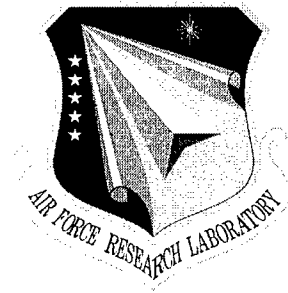


AFRL-IF-RS-TR-1999-269

Final Technical Report

January 2000



**DESIGN, DEVELOPMENT, BENCHMARKING AND
EVALUATION OF PARALLEL APPLICATIONS
FOR HIGH PERFORMANCE EMBEDDED
SYSTEMS**

Syracuse University

Wei-keng Liao, Donald Weiner, and Alok Choudhary

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

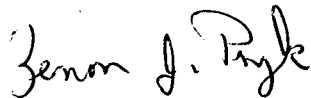
20000308 014

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1999-269 has been reviewed and is approved for publication.

APPROVED:



ZENON J. PRYK
Project Engineer

FOR THE DIRECTOR:



NORTHROP FOWLER
Technical Advisor
Information Technology Division

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTS, 26 Electronic Pky, Rome, NY 13441-4514. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE JANUARY 2000		3. REPORT TYPE AND DATES COVERED Final Nov 96 - May 99
4. TITLE AND SUBTITLE DESIGN, DEVELOPMENT, BENCHMARKING AND EVALUATION OF PARALLEL APPLICATIONS FOR HIGH PERFORMANCE EMBEDDED SYSTEMS			5. FUNDING NUMBERS C - F30602-97-C-0026 PE - 63755D PR - HPCM TA - 00 WU - P1	
6. AUTHOR(S) Wei-keng Liao, Donald Weiner, and Alok Choudhary				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Syracuse University Office of Sponsored Programs 113 Browne Hall Syracuse New York 13441-4514			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFTC 26 Electronic Pky Rome New York 1344-4514			10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-1999-269	
11. SUPPLEMENTARY NOTES Air Force Research Laboratory Project Engineer: Zenon J. Pryk/IFTC/(315) 330-2596				
12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Due to the nature of the algorithms typically employed in applications such as STAP, sensor data fusion, and target detection, it was decided to integrate the signal processing areas of space-time adaptive processing and signal detection. In particular, the following algorithms were parallelized: 1) AFRL (Rome) version of a PEI-staggered post-Doppler STAP algorithm. This algorithm, comprised of more than 23,000 lines of code, included the steps of a) Doppler filter processing, b) weight computation, c) beam forming, d) pulse compression, and e) constant false alarm rate (CFAR) processing. 2) Ozturk (clutter characterization) algorithm. This algorithm is used to analyze random data and includes the steps of a) goodness-of-fit test and b) probability distribution approximation. 3) Ordered-statistic CFAR algorithm. This CFAR algorithm is in addition to the cell averaging CFAR algorithm contained in the PRI-staggered post-Doppler STAP algorithm. In carrying out the algorithm parallelizations, the following task/technical requirements were accomplished: 1) Efficient techniques for high-speed, high-volume I/O applicable to embedded high-performance systems were designed and implemented. 2) Data distribution and redistribution strategies for both inter-task and intra-task data communications in real-time pipelined and parallelized applications were designed and implemented. 3) A documented beta code release was implemented to illustrate the full system with all major functional, technical, programming, documentation, installation, and user application features to be included in the full delivery. 4) The individual algorithms, as well as the integrated applications, were implemented, demonstrated, benchmarked, and evaluated on the Intel Parago and, IBM SP2.				
14. SUBJECT TERMS Signal Processing, High Performance Computing, Programming			15. NUMBER OF PAGES	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

1.0	Background.....	1
2.0	Objectives.....	1
3.0	Administrative Details.....	1
4.0	Participants.....	2
5.0	Accomplishments.....	2
Appendix A.....		A-1
Appendix B.....		B-1
Appendix C.....		C-1
Appendix D.....		D-1
Appendix E.....		E-1

FINAL REPORT
FOR
DESIGN, DEVELOPMENT, BENCHMARKING AND EVALUATION
OF PARALLEL APPLICATIONS
FOR HIGH PERFORMANCE EMBEDDED SYSTEMS

1.0 Background

High performance computing is coming into the mainstream due to progress made in both hardware as well as software support in the past few years. For DoD applications, in particular, the trend toward leveraging off-the-shelf components and systems creates the need to address many system issues relevant to the DoD applications that were largely not considered when high performance computing was used mainly for scientific applications.

These DoD specific issues arise from the particular functional requirements of the intended applications, the frequency requirements for high-speed high-volume data input and output, and real-time requirements for achieving specified throughput and latency. For benchmarking and evaluation of software systems, it is not just sufficient to compute the total execution time of an application, but it is extremely important to study the performance of individual components of an application, the overheads stemming from interactions among the component tasks (e.g. data flow), and the overall performance of an integrated system in terms of the achievable latency and throughput.

2.0 Objectives

The objectives of this effort were to: (a) design, develop and implement individual parallel and portable algorithms plus integrated algorithm systems for applications such as Space-Time Adaptive Processing (STAP), sensor data fusion, and target detection; (b) design and implement efficient Input/Output (I/O), data redistribution and task assignment techniques for embedded high-performance system applications; (c) implement and benchmark the algorithms individually and in integrated applications in the Intel Paragon and demonstrate the performance levels achieved; (d) deliver high-quality software for distribution to DoD researchers nationwide.

3.0 Administrative Details

The sponsor of this effort was the Information Directorate of the Air Force Research Laboratory (AFRL/IF) located in Rome, NY. Funding in the amount of \$359,748 was provided as part of the Common HPC Software Support Initiative (CHSSI) under the DoD High Performance Computing Modernization Program (HPCMP). The duration of

the effort was approximately 29 months, with a start date of December 24, 1996 and an end date of May 15, 1999. Syracuse University was the principal contractor while Northwestern University was a subcontractor.

4.0 Participants

The principal investigators were Drs. Pramod Varshney and Donald Weiner of Syracuse University and Drs. Alok Choudhary and Nagaraj Shenoy of Northwestern University. They were assisted by doctoral students Wei-keng Liao of Syracuse University and Xiaohui Shen of Northwestern University.

Valuable contributions were made by several AFRL (Rome) personnel. Russ Brown, Mike Little, Mark Linderman and Richard Linderman clearly explained their rationale for the changes they had implemented in the STAP algorithm chosen for parallelization. Charles Pedersen and Zen Pryk provided valuable guidance with the CHSSI and HPCMP documentation requirements. In addition, Zen Pryk assisted with the alpha and beta testing and made a major contribution to parallelization of the Ozturk algorithm by converting its FORTRAN code from an interactive to batch mode. Zen, also, removed nonstandard FORTRAN features so that the parallelized version of the Ozturk algorithm could be compiled and run on a variety of high-performance computers.

5.0 Accomplishments

Based upon Government review of our suggestions with regard to algorithms typically employed in applications such as STAP, sensor data fusion, and target detection, it was decided to integrate the signal processing areas of space-time adaptive processing and signal detection. In particular, the following algorithms were parallelized:

- 1) AFRL (Rome) version of a PRI-staggered post-Doppler STAP algorithm. This algorithm, comprised of more than 23,000 lines of code, included the steps of a) Doppler filter processing, b) weight computation, c) beam forming, d) pulse compression, and e) constant false alarm rate (CFAR) processing.
- 2) Ozturk algorithm. This algorithm is used to analyze random data and includes the steps of a) goodness-of-fit test and b) probability distribution approximation.
- 3) Ordered-statistic CFAR algorithm. This CFAR algorithm is in addition to the cell averaging CFAR algorithm contained in the PRI-staggered post-Doppler STAP algorithm.

In carrying out the algorithm parallelizations, the following task/technical requirements were accomplished:

- 1) Efficient techniques for high-speed, high-volume I/O applicable to embedded high-performance systems were designed and implemented.

- 2) Data distribution and redistribution strategies for both inter-task and intra-task data communications in real-time pipelined and parallelized applications were designed and implemented.
- 3) Task assignment and scheduling techniques which can be used to satisfy latency and throughput requirements for high-performance embedded systems were designed and implemented.
- 4) A documented alpha code release was implemented in accordance with the contract schedule using algorithms that provide a representative example of all major technical, programming, documentation, installation and user application features planned for the full delivery.
- 5) A documented beta code release was implemented to illustrate the full system with all major functional, technical, programming, documentation, installation, and user application features to be included in the full delivery.
- 6) The individual algorithms, as well as the integrated applications, were implemented, demonstrated, benchmarked, and evaluated on the Intel Paragon at AFRL (Rome). The performance and optimization levels achieved were demonstrated and the final release delivered to AFRL (Rome).
- 7) A Software System Design Plan that presented prioritized and sequenced timelines for design, development, benchmarking, evaluation and documentation for the individual algorithms and applications chosen for parallelization was documented. Targeted levels of completion and functionality for the alpha, beta, and final code releases, and the format and planned content for the Application Programming Interface were included.
- 8) All computer software developed, assembled, and acquired was delivered to the Government in accordance with its specifications.

Details of the work accomplished are documented in the publications, reports, and manuals included in the appendices attached to this report. These are itemized below:

- 1) Papers presented at conferences (Appendix A)

Choudhary, A., Liao, W. K., Weiner, D., Varshney, P., Linderman, M., Linderman, R., "Design of Parallel Pipelined STAP on High-Performance Computers", Proc. 1997 DoD High Performance Computing Modernization Program Users Group Meeting, San Diego, CA, June 23-26, 1997.

Choudhary, A., Liao, W. K., Weiner, D., Varshney, P., Linderman, M., Linderman, R., "Design Implementation, and Evaluation of Parallel Pipelined STAP on Parallel Computers", Combined International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, Orlando, Florida, March 30-April 3, 1998.

Choudhary, A., Liao, W. K., Weiner, D., Varshney, P., Linderman, M., Linderman, R., "Design and Implementation of Space-Time Adaptive Processing Application on Parallel Computers", Proc. 1998 DoD High Performance Computing Modernization Program Users Group Meeting, Houston, Texas, June 1-5, 1998.

Liao, W. K., Choudhary, A., Weiner, D., Varshney, P., "Multi-Threaded Design and Implementation of Parallel Pipelined STAP on Parallel Computers", 1999 International Parallel Processing Symposium, Puerto Rico, April 1999.

2) Papers submitted for publication (Appendix B)

Choudhary, A., Liao, W. K., Weiner, D., Varshney, P., Linderman, M., Linderman, R., "Design Implementation, and Evaluation of Parallel Pipelined STAP on Parallel Computers", selected for a special collection of papers on STAP and adaptive arrays to appear in an upcoming issue of the IEEE Transactions on Aerospace and Electronic Systems.

Liao, W. K., Choudhary, A., Weiner, D., Varshney, P., "I/O Implementation, and Evaluation of Parallel Pipelined STAP on Parallel Computers", International Conference on High-Performance Computing (HIPC 99), Calcutta, India, Dec. 17-20, 1999.

3) Ph.D. Dissertation (Appendix C)

Liao, W. K., "Parallel Pipelined Computational Model for Space-Time Adaptive Processing", Syracuse University, June 1999.

4) Report and Users' Manual for Ozturk Algorithm (Appendix D)

5) Users' Manual for STAP (Appendix E)

Appendix A

Papers presented at conferences

Design of Parallel Pipelined STAP on High-Performance Computers

Alok Choudhary
(choudhar@ece.nwu.edu)
ECE Department
Northwestern University
Evanston, IL 60208

Wei-Keng Liao, D. Weiner
and P. Varshney
EECS Department
Syracuse University
Syracuse, NY 13244

R. Linderman and M.
Linderman
Rome Laboratory
Surveillance Directorate
Rome, NY 13441

Abstract

This paper presents preliminary results for our ongoing implementation of parallel pipelined STAP algorithm on high-performance computers. In particular, the paper describes the issues involved in parallelization, our approach to parallelization and initial results on some tasks of the STAP algorithm. Initial results are encouraging and show significant performance benefits from our approach. The results demonstrate the scalability of computations and communication.

1. Introduction

The detection of weak target returns embedded in strong ground clutter, interference, and receiver noise is a primary objective of airborne surveillance phased array radars. Space-time adaptive processing (STAP) refers to 2-dimensional adaptive filtering algorithms which take advantage of differences between the spatial and/or Doppler frequencies of the target versus those of the unwanted components of the received waveform in order to separate the target from the disturbances.

The spatial frequency of a signal is a function of its angle of arrival while its Doppler frequency is a function of a relative radial velocity between the airborne platform and that of the corresponding scatterer or jammer. Unwanted signals are attenuated by using STAP algorithms to place nulls in the 2-dimensional frequency plane with respect to their directions of arrival and/or Doppler frequencies. However, high performance computers are required to meet the STAP computational requirements of real-time applications and to increase the flexibility, affordability, and scalability of radar signal processing systems.

In this paper we discuss our progress in implementing a PRI-staggered post-Doppler STAP algorithm on the Rome Laboratory Intel Paragon machine. The algorithm consists of the following steps: 1) application to the data of window and range correction multipliers, 2) calculation of 128-point FFT's for each PRI stagger and every range and channel, 3) solution of the weight vector for each Doppler bin and range gate, 4) application of the weight vector to the test cell data for each Doppler bin and range gate, 5) pulse compression of the array output data for each Doppler bin and range gate. For our study the data cube for a coherent processing interval (CPI) was assumed to be collected from 16 channels, 128 pulses, and 512 range gates. For the parallel implementation we have designed parallel pipelined collection of tasks, where each task itself is parallel. In this paper we present some preliminary results from this implementation. In Section 2 we present the model of computation. Parallelization issues are discussed in Section 3. Section 4 presents some specific details of STAP implementation and software development. Preliminary results are presented in Section 5.

2. Model of Computation

Figure 1 shows the computational model for the type of applications (e.g., STAP) considered in this work and illustrates the computational characteristics found in these applications. Each pipeline shows a number of tasks applied to a set of inputs. The input to the first task in a pipeline is the input to the rest of the tasks is the output of the previous task. The set of pipelines (shades) illustrates that the entire pipeline of tasks is repeated on subsequent data sets. Each block in the pipeline represents one parallel task. That is, the pipeline is a collection of parallel tasks.

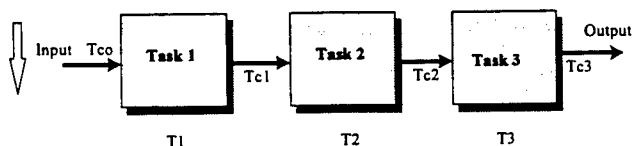


Figure 2: Model of Computation

There exists both spatial and temporal parallelism in such applications. Existence of spatial and temporal parallelism may also result in two types of data dependencies and flow, namely, spatial data dependency and temporal data dependency[1,2]. Intertask data dependency denotes the transfer and reorganization of data to be passed onto the next task in the pipeline. The mode of communication is subtasks of the current tasks to the subtasks of the next task, permitting parallel pipelined communication.

3. PARALLELIZATION ISSUES

Applications such as STAP entail multiple algorithms (or processing steps), each of which performs a particular function, to be executed in a pipelined fashion. Multiple pipelines need to be executed in a staggered manner to satisfy the throughput requirements. Each task needs to be parallelized for the required performance, which in turn requires addressing issues of data distributions on the subset of processors on which a task is parallelized to obtain good efficiency and incur minimal communication overhead. Given that each task is parallelized, data flow among them requires communication among multiple processes of two or more tasks, for which efficient communication scheduling techniques become critical. The problem of input-output of data is another crucial problem and is more challenging in this scenario because data must be redistributed within the pipeline in a timely manner to guarantee the throughput and latency requirements.

3.1 Data Redistribution

In an integrated system which implements several tasks that feed data to each other, data redistribution is required when it is fed from one parallel task to another, or when intermediate results need to be exchanged within a parallel task. This is because the way data is distributed in one task may not be the most appropriate distribution for the task it is supplied to due to algorithmic or efficiency reasons. Furthermore, the number of processors in two communicating tasks may be different because of the required response time from each task and the underlying computations requiring redistribution.

Recently, we developed runtime functions and strategies that perform efficient redistribution of data [4]. These techniques reduce the communication time by minimizing contention on the communication links as well as by minimizing the overhead of processing for redistribution (which adds to the latency of sending messages). We apply lessons from these techniques to implement parallel pipelined STAP application

3.2 Task Scheduling and Assignment

An important factor in the performance of a parallel system, is how the computational load is mapped onto the processors in the system. Ideally, to achieve maximum parallelism, the load must be evenly distributed across the processors. Applications such as STAP employ several algorithms with different computational requirements. Each task must be allocated some processors so that all the tasks can execute concurrently. Furthermore, there is communication among tasks to transfer intermediate data. All these sub-components should be accounted for in making scheduling decisions.

When several parallel tasks need to be executed in a pipelined fashion tradeoffs exist between maximizing throughput and minimizing latency. The throughput requirement says that when allocating processors to tasks, it should be guaranteed that the all input data sets will be handled in a timely manner, that is, the processing rate should not fall behind the input data rate. The response time criteria, on the other hand, require minimizing the latency of computation on a particular set of data input. Clearly, there is a tradeoff. Increasing the throughput invariably means increasing the latency given a fixed set of resources (processors etc.), and vice-versa. In our previous work we have developed techniques for processor allocations to various tasks that balance throughput and latency requirements optimally[6], and these will be used in the STAP algorithm implementation.

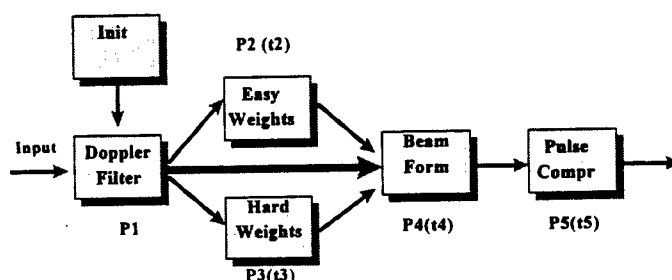


Figure 2: Parallel Pipelined Implementation of STAP

4. Design and Implementation of Parallel Pipelined STAP

Figure 2 shows the design of the parallel pipelined STAP on high-performance computers. There are five basic tasks in addition to initialization. A detailed description of the algorithms can be found in [7,8]. Here we briefly describe each task.

4.1 Parallelization of Steps

The first task is Doppler Filter processing. It involves performing range correction and 128-point FFT. Range correction is done by a windowing operation. The basic parallelization technique in this step is to partition data across the range data, that is, if P_1 processors are allocated to this step, then each processor is responsible for K/P_1 range cells, where K is the number of range cells. Figure 3 illustrates the parallelization of this step.

The second step in this pipeline is computing adaptive weights to be applied to the next CPI. As seen from Figure 2, this computation itself is divided into two parts, namely, "easy" and "hard" Doppler bins. The main difference in the two is the amount of data used and the amount of computation in each of these steps. For each "hard" Doppler bin, the amount of computation is approximately 48 times that in the "easy" Doppler bin. Each of these involves QR factorization. Given the uneven nature of the computations, different sets of processors are allocated to each of these steps, as shown in Figure 2. Note that, as seen from the figure, data computed in the first step needs to be communicated to these two tasks as well as to the task in the third step. The data sent to the third task is more than to the second task (shown by a thick arrow).

The third task (which is actually the second step for the current CPI because the result of the second task is only used in the subsequent time step) is Beamforming. This requires 56 matrix multiplications of 51×32 matrix by a 32×6 matrix. Since the cost of these multiplications can be determined accurately, the computations are equally divided among the allocated

processors for this step. As seen from Figure 2, this step requires data to be communicated from the first as well as second task.

The last step, Pulse compression is performed when the beams are formed. It involves convolution of the received signal with a replica of transmit pulse wave form. This is accomplished by first performing FFTs in of the two inputs, point-wise multiplication of the intermediate result and then computing the inverse FFT. Again, each of these FFTs could be performed on a individual processor, each processor in this task getting equal amount of computation. For more details of the these algorithms, please refer to [5,6].

4.2 Software Development

All the parallel programs development and their integration is being performed using C language and message passing interface (MPI) [3]. All the functions needed for data redistribution etc. are also being developed in the same fashion. This permits easy portability across various platforms which support C language and MPI. Since MPI is becoming a de facto standard for high-performance systems, we believe the software will be portable. To facilitate upward or downward scalability, the number of processors, data sizes and other important parameters are runtime inputs so that the same program can be run on different number of processors without compiling it again. This allows, for example, the same function to be executed on 2, 4 and so on, number of processors.

5. Preliminary Results

The first implementation of techniques and application is being done on the Intel Paragon installed at Rome Laboratories. Due to lack of space only results from the first task and part of the second task are presented. Communication performance, and performance of other tasks will be shown in the presentation at the meeting. Figure 4 shows the performance results for Task 1 (Doppler processing) as a function of processors on the Intel Paragon. Parameters for this task are : Number of range cells = 512, Number of Channels = 16 and number of pulses = 128. For each range cell and channel pair, two 128-point FFTs were performed. The performance results include all the overhead incurred including dynamic memory allocation, windowing and computing staggered inputs. As can be observed, we have obtained linear speedups. Although the performance results are shown only up to 32 processors, we have obtained linear speedups for larger number of processors. On 32 processors, the first step can be performed in approximately 90 milliseconds.

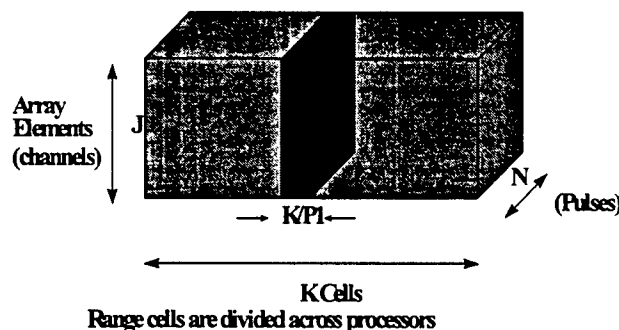


Figure 3: Partitioning Strategy for Step1: Doppler Filter Parallel Task

Figure 5 shows initial result on implementing hard doppler bin task for up to 56 processors. Despite the fact that no optimizations have been incorporated yet, we obtain almost linear speedups. For the 56 processor case, it takes approximately 160 milliseconds for computing hard doppler bins. Easy doppler bins task takes 80 milliseconds on 12 processors. In the presentation, we intend to provide more detailed results.

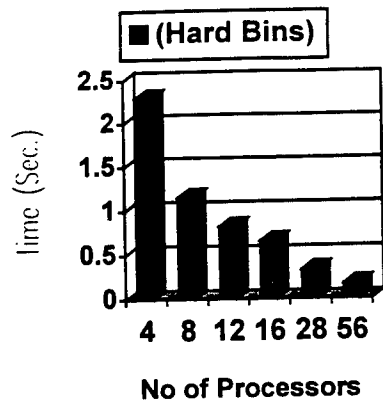


Figure 5

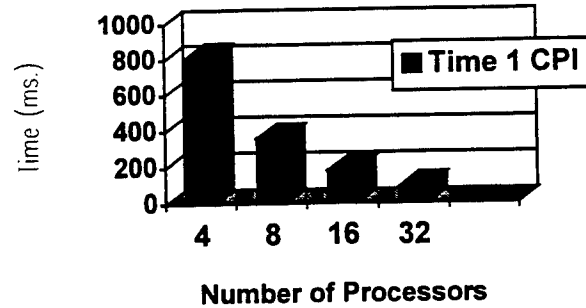


Figure 4

Figure 4: Performance of Doppler Processing (Task 1) as a function of Number of Processors

Figure 5: Performance of Hard Bin (Task 2) as a function of Number of Processors (unoptimized)

Figure 6 shows communication scaling for data transfer from Task 1 to Task 2. In this three-dimensional chart, the two horizontal axes show the number of processors in each task. The vertical axis shows the amount of time for communicating data including all the send-receive overhead for that pair of number of processors in each task. It is clear that there is tremendous scaling in performance of communicating data as the number of processors is increased. This is because the amount of processing for communication per processor is decreased (as it handles less amount of data), amount of data per processor to be communicated is decreased and traffic on links going in and out of each processor is reduced. This is clearly a scalable model and approach for computation and communication.

6. Summary

In this paper we presented initial results in implementing a PRI-staggered post-Doppler STAP algorithm on the Rome Laboratory Intel Paragon machine. The initial results indicate that our approach of parallel pipelined implementation scales well both in terms of communication and computation.

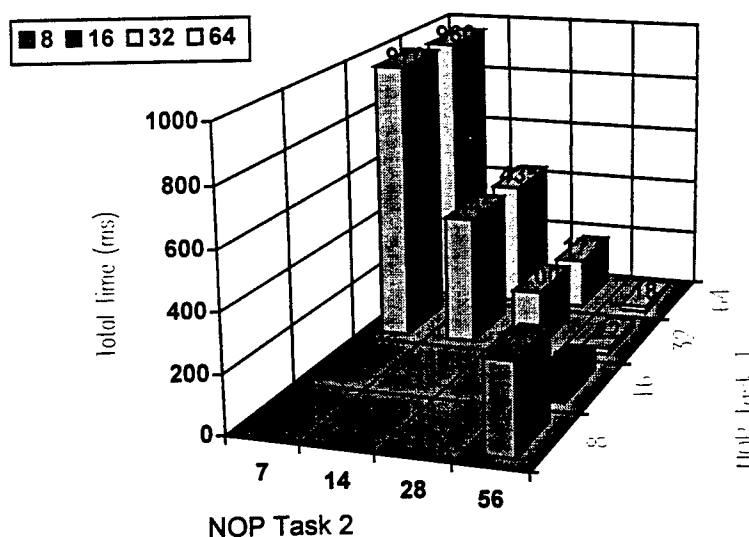


Figure 6: Communication Scaling from Task 1 to Task 2.

Acknowledgements

This work was supported by Air Force Materials Command under contract F30602-97-C-0026. We acknowledge the use of the Intel Paragon at CALTECH for initial development.

References

- [1] A. Choudhary and Ponnusamy, "Parallel Implementation and Evaluation of a Motion Estimation System Algorithm using Several Data Decomposition Strategies", Jour. of Par. and Dist. Comp., Jan. 1992.
- [2] A. N. Choudhary, "Parallel architectures and parallel algorithms for integrated vision systems", Kluwer Academic Publisher, Boston, MA, 1990
- [3] M. Snir et. al., "MPI The Complete Reference", The MIT Press, 1995.
- [4] R. Thakur, A. Choudhary and J. Ramanujam, "Efficient Algorithms for Array Redistribution, IEEE Trans. on Parallel and Dist. Systems", 1995.
- [5] R. Bordawaker, A. Choudhary and J. M. del Rosario, "Runtime Primitives for Parallel I/O, Supercomputing '93", November 1993, Portland, OR.
- [6] A. N. Choudhary, B. Narahari, D. M. Nicol and R. Simha, "Optimal Processor Assignment for Pipeline Computations", IEEE Trans. on Par. and Dist. Systems, April, 94.
- [7] M. Linderman and R. Linderman, "Real-Time STAP Demonstration on an embedded high-performance computer," 1997 National Radar Conference.
- [8] R. Brown and R. Linderman, "Algorithm Development for an Airborne Real-Time STAP Demonstration," 1997 National Radar Conference

Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers

Alok Choudhary

ECE Department
Northwestern University
Evanston, IL 60208
email: choudhar@ece.nwu.edu

Wei-keng Liao,
Donald Weiner, and
Pramod Varshney

EECS Department
Syracuse University
Syracuse, NY 13244

Richard Linderman and
Mark Linderman

Air Force Research Laboratory
Information Directorate
Rome, NY 13441

Abstract

This paper presents performance results for the design and implementation of parallel pipelined Space-Time Adaptive Processing (STAP) algorithms on parallel computers. In particular, the paper describes the issues involved in parallelization, our approach to parallelization and performance results on an Intel Paragon. The paper also discusses the process of developing software for such an application on parallel computers when latency and throughput are both considered together and presents tradeoffs considered with respect to inter and intra-task communication and data redistribution. The results show that not only scalable performance was achieved for individual component tasks of STAP but linear speedups were obtained for the integrated task performance, both for latency as well as throughput. Results are presented for up to 236 compute nodes (limited by the machine size available to us). Another interesting observation made from the implementation results is that performance improvement due to the assignment of additional processors to one task can improve the performance of other tasks without any increase in the number of processors assigned to them. Normally, this cannot be predicted by theoretical analysis.

1 Introduction

Space-time adaptive processing (STAP) is a well known technique in the area of airborne surveillance radars, which is used to detect weak target returns embedded in strong ground clutter, interference, and receiver noise. Data processing for STAP refers to a 2-dimensional adaptive filtering algorithm which attenuates unwanted signals by placing nulls in the frequency domain with respect to their directions of arrival and/or Doppler frequencies. Most STAP

applications consume great amounts of computational resources and are also required to operate in real time. High performance computers are becoming mainstream due to the progress made in hardware as well as software support in the last few years. They can satisfy the STAP computational requirements of real-time applications while increasing the flexibility, affordability, and scalability of radar signal processing systems. However, efficient parallelization of STAP, which consists of several different algorithms is challenging, and requires several optimizations.

This paper describes our parallel pipelined implementation of a PRI-staggered post-Doppler STAP algorithm. The design and implementation of the application is portable. Performance results are presented for the Intel Paragon at the Air Force Research Laboratory (AFRL), Rome, New York. AFRL has successfully implemented this STAP algorithm onboard an airborne platform and performed four flight experiments in May and June 1996 [8]. In that real-time demonstration, live data from a phased array radar was processed by Intel Paragon machine and results showed that high performance computers can deliver a significant performance gain. However, that implementation only used compute nodes of the machine as independent resources in a round robin fashion to run different instances of STAP (rather than speeding up one instance of STAP.) Using this approach, the throughput may be improved, but the latency is limited by what can be achieved using one compute node. The algorithm consists of the following steps: 1) Doppler filter processing, 2) weight computation, 3) beamforming, 4) pulse compression, and 5) CFAR processing.

For our parallel implementation of this real application we have designed a model of parallel pipeline system where each pipeline is a collection of tasks and each task itself is parallelized. This parallel pipeline model was applied to the STAP algorithm with each step as a task in a pipeline. This permits us to significantly improve latency as well as

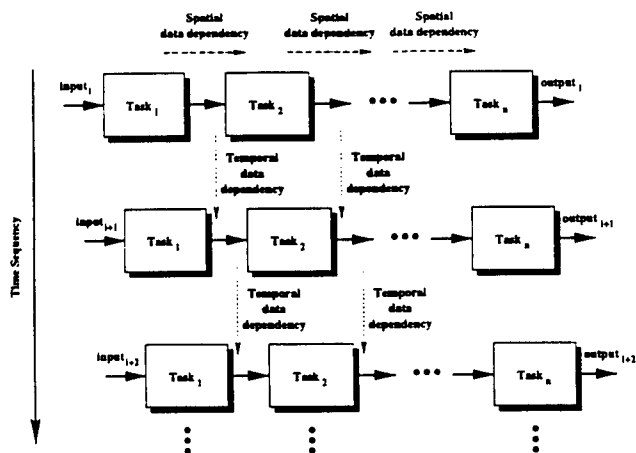


Figure 1. Model of the parallel pipeline system. (Note that $Task_i$ for all input instances is executed on the same number of processors.)

throughput. In this paper we present results from this implementation. Furthermore, we present the process of parallelization and software design considerations including those for portability, task mapping, parallel data redistribution, parallel pipelining and issues involving in measuring performance in implementations when not only the performance of individual tasks is important, but overall performance of the integrated system is critical. We demonstrate the performance and scalability for a large number of processors.

The rest of the paper is organized as follows: in Section 2, we present the parallel pipeline system model and discuss some parallelization issues and approaches for implementation of STAP algorithms. Section 3 presents the implementation. Performance results and conclusions are presented in Section 4 and Section 5 respectively.

2 Model of the parallel pipeline system

The system model for the type of STAP applications considered in this work is shown in Figure 1. This model is suitable for the computational characteristics found in these applications. A pipeline is a collection of tasks which are executed sequentially. The input to the first task is obtained normally from sensors or other input devices and the inputs to the rest of the tasks in the pipeline are the outputs of their previous tasks. The set of pipelines shown in the figure indicates that the same pipeline is repeated on subsequent input data sets. Each block in a pipeline represents one parallel task, which itself is parallelized on multiple (different number of) processors.

In such a system, there exist both spatial and temporal parallelism that result in two types of data dependencies and

flows, namely, spatial data dependency and temporal data dependency [4, 6]. Spatial data dependency can be classified into inter-task data dependency and intra-task data dependency. Intra-task data dependencies arise when a set of subtasks needs to exchange intermediate results during the execution of a parallel task in a pipeline. Inter-task data dependency is due to the transfer and reorganization of data passed onto the next parallel task in the pipeline. Temporal data dependency occurs when some form of output generated by the tasks executed on the previous data set are needed by tasks executing the current data set. We will later see that STAP has both types of data dependencies.

2.1 Parallelization issues and approaches

Applications such as STAP entail multiple algorithms (or processing steps), each of which performs particular functions, to be executed in a pipelined fashion. Each task needs to be parallelized for the required performance, which, in turn, requires addressing the issue of data distribution on the subset of processors on which a task is parallelized to obtain good efficiency and incur minimal communication overhead.

2.1.1 Inter-task data redistribution

In an integrated system which implements several tasks that feed data to each other, data redistribution is required when it is fed from one parallel task to another. This is because the way data distributed in one task may not be the most appropriate distribution for another task for algorithmic or efficiency reasons. Data redistribution also allows concentration of communication at the beginning and the end of each task. We have developed runtime functions and strategies that perform efficient data redistribution [10]. These techniques reduce the communication time by minimizing contention on the communication links as well as by minimizing the overhead of processing for redistribution (which adds to the latency of sending messages). We take advantage of lessons learned from these techniques to implement the parallel pipelined STAP application.

2.1.2 Task scheduling and processor assignment

An important factor in the performance of a parallel system, is how the computational load is mapped onto the processors in the system. Ideally, to achieve maximum parallelism, the load must be evenly distributed across the processors. The problem of statically mapping the workload of a parallel algorithm to processors in a distributed memory system, has been studied under different problem models, such as [1, 2]. These static mapping policies do not model applications consisting of a sequence of tasks (algorithms),

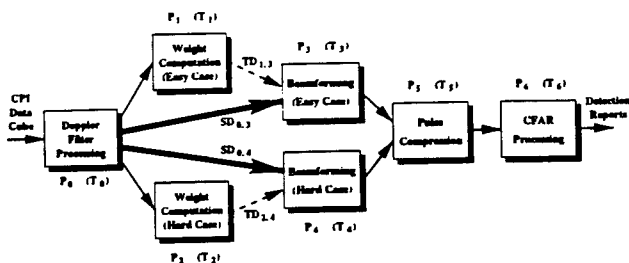


Figure 2. Implementation of parallel pipelined STAP. Arrows connecting task blocks represent data transfer between tasks.

where the output of one task becomes the input to the next task in the sequence.

Optimal use of resources is particularly important in high-performance embedded applications due to limited resources and other constraints such as desired latency or throughput [5]. When several parallel tasks need to be executed in a pipelined fashion, tradeoffs exist between assigning processors to maximize the overall throughput and assigning processors to minimize a single data set's response time (or latency.)

3 Design and implementation

The design of the parallel pipelined STAP algorithm is shown in Figure 2. The parallel pipeline system consists of seven basic tasks. We refer to the parallel pipeline as simply a pipeline in the rest of this paper. Both the weight computation and the beamforming tasks are divided into two parts, namely, "easy" and "hard" Doppler bins. The hard Doppler bins are those in which significant ground clutter is expected and the remaining bins are easy Doppler bins. The main difference between the two is the amount of data used and the amount of computation required. The input data set for the pipeline is obtained from a phased array radar and is formed in terms of a coherent processing interval (CPI). Each CPI data set is a 3-dimensional complex data cube. The output of the pipeline is a report on the detection of possible targets. Each task i , $0 \leq i < 7$, is parallelized by evenly partitioning its work load among P_i processors. The execution time associated with task i , T_i , consists of the time to receive data from the previous task, computation time, and time to send results to the next task.

For the computation of the weight vectors for the current CPI data cube, data cubes from previous CPIs are used as input data. This introduces temporal data dependency. Temporal data dependencies are represented by arrows with dashed lines, $TD_{1,3}$ and $TD_{2,4}$, in Figure 2 where $TD_{i,j}$ represents temporal data dependency of task j on data from task i . In a similar manner, spatial data dependencies $SD_{i,j}$

can be defined and are indicated in Figure 2 by arrows with solid lines.

Throughput and latency are two important measures for performance evaluation on a pipeline system. The throughput of our pipeline system is the inverse of the maximum execution time among all tasks. The latency of this pipeline system is the time between the arrival of the CPI data cube at the system input and the time at which the detection report is available at the system output.

$$throughput = \frac{1}{\max_{0 \leq i \leq 6} T_i} \quad (1)$$

$$latency = T_0 + \max_{i=3,4} T_i + T_5 + T_6. \quad (2)$$

The temporal data dependency does not affect the latency because weight computation tasks use data from the previous instance rather than current CPI. The filtered CPI data cube sent to the beamforming task does not wait for the completion of its weight computation. This explains why equation (2) does not contain T_1 and T_2 . A detailed description of the STAP algorithm we used can be found in [3, 7].

4 Performance results

The implementation of the STAP application based on our parallel pipeline system model was done on the Intel Paragon at the Air Force Research Laboratory, Rome, New York. All the parallel programs development and their integration was performed using C language and message passing interface (MPI) [9]. This permits easy portability across various platforms which support C language and MPI. In our implementation, asynchronous send and receive function calls were used in order to overlap communication and computation.

4.1 Computation costs

The task of computing hard weights is the most computationally demanding task. The Doppler filter processing task is the second most demanding task. Naturally, more processors are assigned to these two tasks in order to obtain a good performance. For each task in the STAP algorithm, parallelization was done by evenly dividing computational load across processors. Figure 3 gives the computation performance results as functions of numbers of processors and the corresponding speedup on the AFRL Intel Paragon. For each task, we obtained linear speedups.

4.2 Inter-task communication

Inter-task communication refers to the communication between sending and receiving (distinct and parallel) tasks.

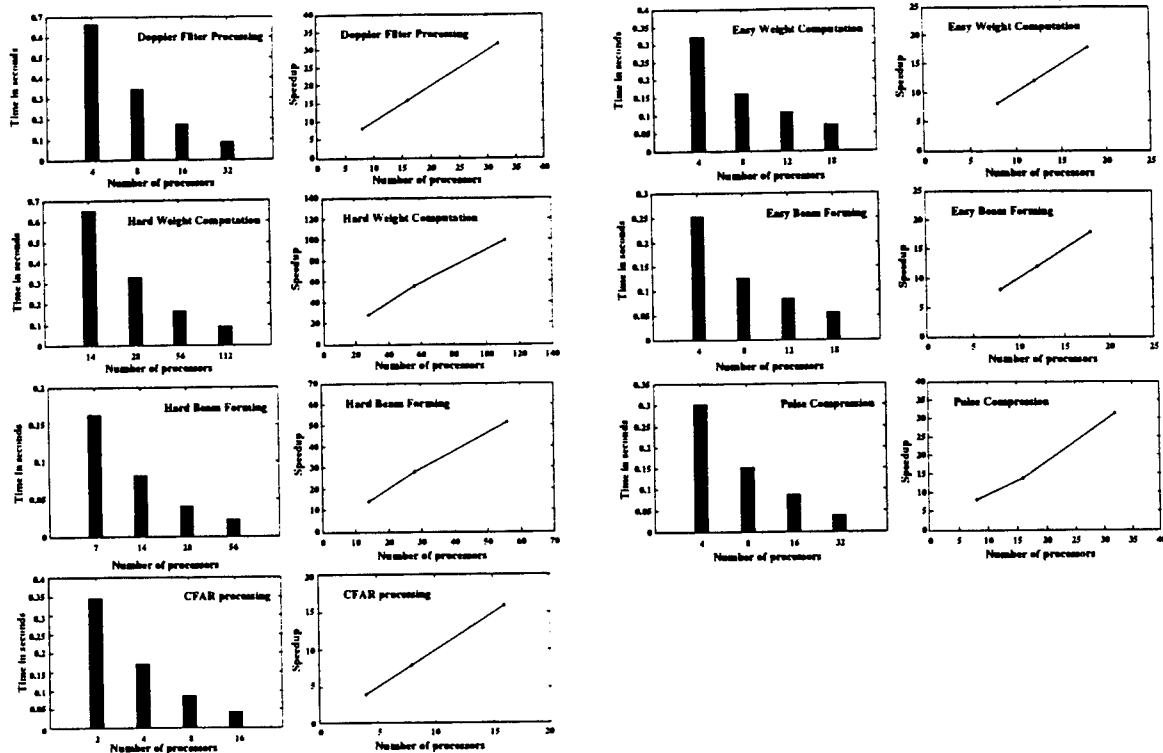


Figure 3. Performance of computation as a function of number of processors.

Table 1. Timing results of inter-task communication. Time in seconds. # proc: number of processors.

	# proc	easy weight		hard weight				easy BF		hard BF	
		16		56		112		16		16	
		send	recv	send	recv	send	recv	send	recv	send	recv
Doppler filter	8	.1332	.4339	.1332	.3603	.1332	.4441	.1332	.4509	.1332	.4395
	16	.0679	.1780	.0679	.1048	.0679	.1837	.0679	.1955	.0679	.1843
	32	.0340	.0511	.0332	.0034	.0340	.0563	.0340	.0646	.0340	.0519

	# proc	easy beamforming			
		8		16	
		send	recv	send	recv
easy weight	4	.0005	.1956	.0007	.2570
	8	.0088	.0883	.0004	.0905
	16	.0768	.0807	.0003	.0660

	# proc	hard beamforming			
		8		16	
		send	recv	send	recv
hard weight	28	.0007	.1798	.0007	.2485
	56	.0100	.1468	.0065	.0765
	112	.1824	.1398	.0005	.0543

	# proc	pulse compression			
		8		16	
		send	recv	send	recv
easy BF	4	.0069	.5016	.0069	.5714
	8	.0036	.1379	.0036	.2090
	16	.0580	.0771	.0022	.0569
hard BF	4	.0054	.5016	.0054	.5714
	8	.0029	.1379	.0030	.2090
	16	.1159	.0771	.0017	.0569

	#proc	CFAR processing			
		4		8	
		send	recv	send	recv
pulse compression	4	.0099	.3351	.0098	.3348
	8	.0053	.0662	.0051	.1750
	16	.1256	.0435	.0028	.1783

This communication cost depends on both processor assignment for each task as well as on the volume and extent of data reorganization. Table 1 presents the inter-task communication timing results. Each sub-table considers pairs of tasks where the number of processors (# proc) for both tasks are varied. In some cases timing results shown in the tables

contain idle time for waiting for the corresponding task to complete. This happens when receiving task's computation part completes before the sending task has generated data to send.

From most of the results the following important observations can be made. First, when the number of processors

Table 2. Performance results for 3 cases with different processor assignments.

case 1: total number of processors = 236					
	# proc	recv	comp	send	total
Doppler filter	32	.0055	.0874	.0348	.1276
easy weight	16	.0493	.0913	.0003	.1408
hard weight	112	.0555	.0831	.0005	.1390
easy BF	16	.0658	.0708	.0021	.1387
hard BF	28	.0936	.0414	.0010	.1361
pulse compression	16	.0551	.0776	.0028	.1355
CFAR	16	.0910	.0434	-	.1344
throughput		7.2659			
latency		0.3622			

case 2: total number of processors = 118					
	# proc	recv	comp	send	total
Doppler filter	16	.0110	.1714	.0668	.2492
easy weight	8	.0998	.1636	.0003	.2637
hard weight	56	.0979	.1636	.0005	.2621
easy BF	8	.1302	.1267	.0036	.2605
hard BF	14	.1782	.0822	.0017	.2622
pulse compression	8	.1027	.1543	.0051	.2621
CFAR	8	.1742	.0864	-	.2606
throughput		3.7959			
latency		0.6805			

case 3: total number of processors = 59					
	# proc	recv	comp	send	total
Doppler filter	8	.0219	.3509	.1296	.5024
easy weight	4	.1796	.3254	.0003	.5053
hard weight	28	.1779	.3265	.0006	.5050
easy BF	4	.2439	.2529	.0068	.5037
hard BF	7	.3370	.1636	.0032	.5039
pulse compression	4	.1806	.3067	.0097	.4970
CFAR	4	.3240	.1723	-	.4963
throughput		1.9898			
latency		1.3530			

is unbalanced, the communication performance is not very good. Second, as the number of processors is increased in the sending and receiving tasks, communication scales tremendously. This happens for two reasons. One, each processor has less data to reorganize, pack and send and each processor has less data to receive; and two, contention at sending and receiving processors is reduced. Thus, it is not sufficient to improve the computation times for such parallel pipelined applications to improve throughput and latency.

Because of the asynchronous send used in the implementation, the results shown here are visible sending time and the actual sending action may occur in other portions of the task. Similar to the receiving time, sending time may also contain waiting time for the completion of sending requests in the previous loop. With large number of processors, there is tremendous scaling in performance of communicating data as the number of processors is increased. This is because the amount of processing for communication per processor is decreased (as it handles less amount of data), amount of data per processor to be communicated is decreased and traffic on links going in and out of each

Table 3. Performance results for adding 4 more processors to case 2 in Table 2.

total number of processors = 122					
	# proc	recv	comp	send	total
Doppler filter	20	.0090	.1395	.0540	.2024
easy weight	8	.0519	.1633	.0003	.2155
hard weight	56	.0486	.1644	.0005	.2135
easy BF	8	.0815	.1272	.0037	.2124
hard BF	14	.1232	.0823	.0018	.2073
pulse compression	8	.0519	.1543	.0051	.2113
CFAR	8	.1240	.0864	-	.2105
throughput		5.0213			
latency		0.5498			

Table 4. Performance results for adding 16 more processors to the case in Table 3.

total number of processors = 138					
	# proc	recv	comp	send	total
Doppler filter	20	.0091	.1395	.0541	.2027
easy weight	8	.0516	.1633	.0003	.2152
hard weight	56	.0488	.1644	.0005	.2137
easy BF	8	.0819	.1273	.0037	.2129
hard BF	14	.1301	.0823	.0018	.2142
pulse compression	16	.1337	.0775	.0028	.2140
CFAR	16	.1701	.0434	-	.2135
throughput		4.9052			
latency		0.4247			

processor is reduced. This model scales well for both computation and communication.

4.3 Integrated system performance

Integrated system refers to the evaluation of performance when all the tasks are considered together. Throughput (CPIs per second) and latency (seconds per CPI) are the two most important measures for performance evaluation in addition to individual task computation time and inter-task communication time. Table 2 gives timing results for three different cases with different processor assignments. From these 3 cases, it is clear that even for latency and throughput measures we obtain linear speedups from our experiments. Given that this scale up is up to 236 processors (we were limited to these number of processors due to the size of the machine), we believe these are very good results.

As discussed in section 2, tradeoffs exist between assigning processors to maximize throughput and to minimize latency, given limited resources. Using two examples, we illustrate how further performance improvements may (or may not) be achieved if few extra processors are available. We now take case 2 from Table 2 as an example and add some extra processors to tasks to analyze its affect to the throughput and latency. Suppose that case 2 has fulfilled the minimum throughput requirement and more processors

can be added. Table 3 shows that adding 4 more processors to Doppler filter processing task not only increases the throughput but also reduces the latency. This is because the communication amount for each send and receive between Doppler filter processing task to weight computation and to beamforming tasks is reduced (Table 3). So, clearly adding processors to one task not only affects that task's performance but has a measurable effect on the performance of other tasks. By increasing the number of processors 3%, the improvement in throughput is 32% and in latency is 19%. *Such effects are very difficult to capture in purely theoretical models because of the secondary effects.*

Since parallel computation load may be different among tasks, bottleneck problems arise when some tasks in the pipeline do not have proper numbers of processors assigned. If the number of processors assigned to one task with heavy work load is not enough to catch up the input data rate, this task becomes a bottleneck in the pipeline system. Hence, it is important to maintain approximately the same computation time among tasks in the pipeline system to maximize the throughput and also achieve higher processor utilization. One bottleneck task can be seen when its computation time is relatively much larger than the rest of the tasks. The entire system's performance degrades because the rest of the tasks have to wait for bottleneck task's completion to send/receive data to/from it no matter how many more processors assigned to them and how fast they can complete their jobs. Therefore, poor task scheduling and processor assignment will cause significant portion of idle time in the resulted communication costs. In Table 4 we added a total of 16 more processors to pulse compression and CFAR processing tasks to the case in Table 3. Comparing to case 2 in Table 2, we can see that the throughput increased. However, the throughput did not improve compared to the results in Table 3, even though this assignment has 16 more processors. In this case, the weight tasks are bottleneck tasks because their computation costs are relatively higher than other tasks. We can see that the receiving time of the rest of tasks are much larger than their computation time. A significant portion of idle time waiting for the completion of weight tasks is in the receiving time. On the other hand, we observe 23% improvement in the latency. This is because the computation time is reduced in the last two tasks with more processors assigned. T_5 and T_6 in equation (2) decrease and therefore the latency is reduced.

5 Conclusions

In this paper we presented performance results for a PRI-staggered post-Doppler STAP algorithm implementation on the Intel Paragon machine at Air Force Research Laboratory, Rome, New York. The results indicate that our approach of parallel pipelined implementation scales well

both in terms of communication and computation. For the integrated pipeline system, the throughput and latency also demonstrate the linear scalability of our design. Our design and implementation not only shows tradeoffs in parallelization, processor assignment, and various overheads in inter and intra-task communication etc., but it also shows that accurate performance measurement of these systems is very important. Consideration of issues such as cache performance when data is packed and unpacked, and impact of the parallelization and processor assignment for one task on another task are crucial. This is normally not easily captured in theoretical models. In the future we plan to incorporate further optimizations including multi-threading, multiple pipelines and multiple processors on each compute node.

6 Acknowledgments

This work was supported by Air Force Materials Command under contract F30602-97-C-0026. We acknowledge the use of the Intel Paragon at Caltech for initial development.

References

- [1] M. Berger and S. Bokhari. "A Partitioning Strategy for Nonuniform Problems on Multiprocessors,". *IEEE Trans. on Computers*, 36(5):570-580, May 1987.
- [2] F. Berman and L. Snyder. "On Mapping Parallel Algorithms into Parallel Architectures,". *Journal of Parallel and Distributed Computing*, 4:439-458, 1987.
- [3] R. Brown and R. Linderman. "Algorithm Development for an Airborne Real-Time STAP Demonstration,". *IEEE National Radar Conference*, 1997.
- [4] A. Choudhary. *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems*. Kluwer Academic Publisher, Boston, MA, 1990.
- [5] A. Choudhary, B. Narahari, D. Nicol, and R. Simha. "Optimal Processor Assignment for Pipeline Computations,". *IEEE Trans. on Parallel and Distributed Systems*, Apr. 1994.
- [6] A. Choudhary and R. Ponnusamy. "Parallel Implementation and Evaluation of a Motion Estimation System Algorithm using Several Data Decomposition Strategies,". *Journal of Parallel and Distributed Computing*, Jan. 1992.
- [7] M. Linderman and R. Linderman. "Real-Time STAP Demonstration on an Embedded High Performance Computer,". *IEEE National Radar Conference*, 1997.
- [8] M. Little and W. Berry. "Real-Time MultiChannel Airborne Radar Measurements,". *IEEE National Radar Conference*, 1997.
- [9] M. Snir and et. al. *MPI The Complete Reference*. The MIT Press, 1995.
- [10] R. Thakur, A. Choudhary, and J. Ramanujam. "Efficient Algorithms for Array Redistribution,". *IEEE Trans. on Parallel and Distributed Systems*, 1995.

Design and Implementation of Space-Time Adaptive Processing Application on Parallel Computers

Alok Choudhary*

ECE Department
Northwestern University
Evanston, IL 60208
email: choudhar@ece.nwu.edu

Wei-keng Liao,
Donald Weiner, and
Pramod Varshney

EECS Department
Syracuse University
Syracuse, NY 13244

Richard Linderman and
Mark Linderman

Air Force Research Laboratory
Information Directorate
Rome, NY 13441

Abstract

This paper presents performance results for our ongoing implementation of parallel pipelined Space-Time Adaptive Processing (STAP) algorithms on Intel Paragon at the Air Force Research Laboratory (AFRL), Rome, New York. In particular, the paper describes the issues involved in designing the parallel pipeline computation model on parallel computers. The paper also discusses the process of developing software for STAP applications on parallel computers when latency and throughput are both considered together and presents tradeoffs. The results show that linear speedups were obtained for the integrated task performance, both for latency as well as throughput. Another interesting observation made from the implementation results is that performance improvement due to the assignment of additional processors to one task can improve the performance of other tasks without any increase in the number of processors assigned to them. Normally, this cannot be predicted by theoretical analysis.

1 Introduction

Space-time adaptive processing (STAP) is a well known technique in the area of airborne surveillance radars, which is used to detect weak target returns embedded in strong ground clutter, interference, and receiver noise. Data processing for STAP refers to a 2-dimensional adaptive filtering algorithm which attenuates unwanted signals by placing nulls in the frequency domain with respect to their directions of arrival and/or Doppler frequencies. Most STAP applications consume great amounts of computational resources and are also required to operate in real time. High performance computers are becoming mainstream due to the progress made in hardware as well as software support in the last few years. They can satisfy the STAP computational requirements of real-time applications while increasing the flexibility, affordability, and scalability of radar signal processing systems. However, efficient parallelization of STAP, which consists of several different algorithms is challenging, and requires several optimizations.

This paper describes our parallel pipelined implementation of a PRI-staggered post-Doppler STAP algorithm. The design and implementation of the application is portable. Performance results are presented for the Intel Paragon at the Air Force Research Laboratory (AFRL), Rome, New York. AFRL has successfully implemented this STAP algorithm onboard an airborne platform and performed four flight experiments in May and June 1996 [Little and Berry, 1997]. In that real-time demonstration, live data from a phased array radar was processed by Intel Paragon machine and results showed that high performance computers can deliver a significant performance

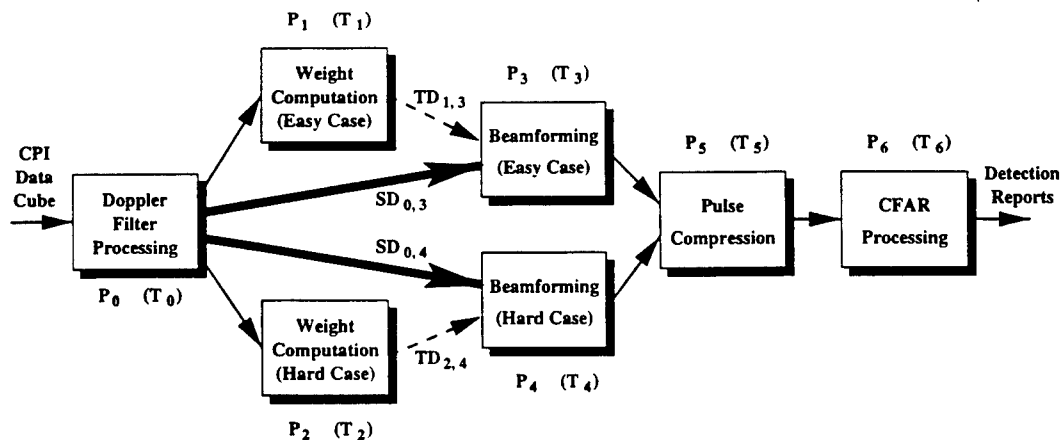


Figure 1. Implementation of parallel pipelined STAP. Arrows connecting task blocks represent data transfer between tasks.

gain. However, that implementation only used compute nodes of the machine as independent resources in a round robin fashion to run different instances of STAP (rather than speeding up one instance of STAP.) Using this approach, the throughput may be improved, but the latency is limited by what can be achieved using one compute node. The algorithm consists of the following steps: 1) Doppler filter processing, 2) weight computation, 3) beamforming, 4) pulse compression, and 5) CFAR processing.

For our parallel implementation of this real application we have designed a model of parallel pipeline system where each pipeline is a collection of tasks and each task itself is parallelized. This parallel pipeline model was applied to the STAP algorithm with each step as a task in a pipeline. This permits us to significantly improve latency as well as throughput. In this paper we present results from this implementation. Furthermore, we present the process of parallelization and issues involving in measuring performance in implementations when not only the performance of individual tasks is important, but overall performance of the integrated system is critical. We demonstrate the performance and scalability for a large number of processors.

The rest of the paper is organized as follows: Section 2 presents the design and implementation. Performance results and conclusions are given in Section 3 and Section 4 respectively.

2 Design and Implementation

The design of the parallel pipelined STAP algorithm is shown in Figure 1. The parallel pipeline system consists of seven basic tasks. We refer to the parallel pipeline as simply a pipeline in the rest of this paper. The input data set for the pipeline is obtained from a phased array radar and is formed in terms of a coherent processing interval (CPI). Each CPI data set is a 3-dimensional complex data cube. The output of the pipeline is a report on the detection of possible targets. Each task i , $0 \leq i < 7$, is parallelized by evenly partitioning its work load among P_i processors. The execution time associated with task i , T_i , consists of the time to receive data from the previous task, computation time, and time to send results to the next task.

For the computation of the weight vectors for the current CPI data cube, data cubes from previous CPIs are used as input data. This introduces temporal data dependency. Temporal data dependencies are represented by arrows with dashed lines, $TD_{1,3}$ and $TD_{2,4}$, in Figure 1 where $TD_{i,j}$ represents temporal data dependency of task j on data from task i . In a similar manner, spatial data dependencies $SD_{i,j}$ can be defined and are indicated in Figure 1 by arrows with solid lines.

Throughput and latency are two important measures for performance evaluation on a pipeline system. The

Table 1. Performance results for 3 cases with different processor assignments.

case 1: total number of processors = 236					
	# proc	recv	comp	send	total
Doppler filter	32	.0055	.0874	.0348	.1276
easy weight	16	.0493	.0913	.0003	.1408
hard weight	112	.0555	.0831	.0005	.1390
easy BF	16	.0658	.0708	.0021	.1387
hard BF	28	.0936	.0414	.0010	.1361
pulse compression	16	.0551	.0776	.0028	.1355
CFAR	16	.0910	.0434	-	.1344
throughput	7.2659				
latency	0.3622				

case 2: total number of processors = 118					
	# proc	recv	comp	send	total
Doppler filter	16	.0110	.1714	.0668	.2492
easy weight	8	.0998	.1636	.0003	.2637
hard weight	56	.0979	.1636	.0005	.2621
easy BF	8	.1302	.1267	.0036	.2605
hard BF	14	.1782	.0822	.0017	.2622
pulse compression	8	.1027	.1543	.0051	.2621
CFAR	8	.1742	.0864	-	.2606
throughput	3.7959				
latency	0.6805				

case 3: total number of processors = 59					
	# proc	recv	comp	send	total
Doppler filter	8	.0219	.3509	.1296	.5024
easy weight	4	.1796	.3254	.0003	.5053
hard weight	28	.1779	.3265	.0006	.5050
easy BF	4	.2439	.2529	.0068	.5037
hard BF	7	.3370	.1636	.0032	.5039
pulse compression	4	.1806	.3067	.0097	.4970
CFAR	4	.3240	.1723	-	.4963
throughput	1.9898				
latency	1.3530				

throughput of our pipeline system is the inverse of the maximum execution time among all tasks. The latency of this pipeline system is the time between the arrival of the CPI data cube at the system input and the time at which the detection report is available at the system output.

$$throughput = \frac{1}{\max_{0 \leq i \leq 6} T_i} \quad (1)$$

$$latency = T_0 + \max_{i=3,4} T_i + T_5 + T_6. \quad (2)$$

The temporal data dependency does not affect the latency because weight computation tasks use data from the previous instance rather than current CPI. The filtered CPI data cube sent to the beamforming task does not wait for the completion of its weight computation. This explains why equation (2) does not contain T_1 and T_2 . A detailed description of the STAP algorithm we used can be found in [Brown and Linderman, 1997, Linderman and Linderman, 1997].

3 Performance Results

The implementation of the STAP application based on our parallel pipeline system model was done on the Intel Paragon at the Air Force Research Laboratory, Rome, New York. All the parallel programs development and their integration was performed using C language and message passing interface (MPI) [Snir and et al., 1995]. This permits easy portability across various platforms which support C language and MPI. In our implementation, asynchronous send and receive function calls were used in order to overlap communication and computation.

Table 2. Performance results for adding 4 more processors to case 2 in Table 1.

total number of processors = 122		Time in seconds			
	# proc	recv	comp	send	total
Doppler filter	20	.0090	.1395	.0540	.2024
easy weight	8	.0519	.1633	.0003	.2155
hard weight	56	.0486	.1644	.0005	.2135
easy BF	8	.0815	.1272	.0037	.2124
hard BF	14	.1232	.0823	.0018	.2073
pulse compression	8	.0519	.1543	.0051	.2113
CFAR	8	.1240	.0864	-	.2105
throughput	5.0213				
latency	0.5498				

Table 3. Performance results for adding 16 more processors to the case in Table 2.

total number of processors = 138		Time in seconds			
	# proc	recv	comp	send	total
Doppler filter	20	.0091	.1395	.0541	.2027
easy weight	8	.0516	.1633	.0003	.2152
hard weight	56	.0488	.1644	.0005	.2137
easy BF	8	.0819	.1273	.0037	.2129
hard BF	14	.1301	.0823	.0018	.2142
pulse compression	16	.1337	.0775	.0028	.2140
CFAR	16	.1701	.0434	-	.2135
throughput	4.9052				
latency	0.4247				

3.1 Integrated System Performance

Integrated system refers to the evaluation of performance when all the tasks are considered together. Throughput (CPIs per second) and latency (seconds per CPI) are the two most important measures for performance evaluation in addition to individual task computation time and inter-task communication time. Table 1 gives timing results for three different cases with different processor assignments. From these 3 cases, it is clear that even for latency and throughput measures we obtain linear speedups from our experiments. Given that this scale up is up to 236 processors (we were limited to these number of processors due to the size of the machine), we believe these are very good results.

Tradeoffs exist between assigning processors to maximize throughput and to minimize latency, given limited resources. Using two examples, we illustrate how further performance improvements may (or may not) be achieved if few extra processors are available. We now take case 2 from Table 1 as an example and add some extra processors to tasks to analyze its affect to the throughput and latency. Suppose that case 2 has fulfilled the minimum throughput requirement and more processors can be added. Table 2 shows that adding 4 more processors to Doppler filter processing task not only increases the throughput but also reduces the latency. This is because the communication amount for each send and receive between Doppler filter processing task to weight computation and to beamforming tasks is reduced (Table 2). So, clearly adding processors to one task not only affects that task's performance but has a measurable effect on the performance of other tasks. By increasing the number of processors 3%, the improvement in throughput is 32% and in latency is 19%. *Such effects are very difficult to capture in purely theoretical models because of the secondary effects.*

Since parallel computation load may be different among tasks, bottleneck problems arise when some tasks in the pipeline do not have proper numbers of processors assigned. If the number of processors assigned to one task with heavy work load is not enough to catch up the input data rate, this task becomes a bottleneck in the pipeline system. Hence, it is important to maintain approximately the same computation time among tasks in the pipeline system to maximize the throughput and also achieve higher processor utilization. One bottleneck task can be seen when its computation time is relatively much larger than the rest of the tasks. The entire system's performance degrades because the rest of the tasks have to wait for bottleneck task's completion to send/receive data to/from it no matter how many more processors assigned to them and how fast they can complete their jobs.

Therefore, poor task scheduling and processor assignment will cause significant portion of idle time in the resulted communication costs. In Table 3 we added a total of 16 more processors to pulse compression and CFAR processing tasks to the case in Table 2. Comparing to case 2 in Table 1, we can see that the throughput increased. However, the throughput did not improve compared to the results in Table 2, even though this assignment has 16 more processors. In this case, the weight tasks are bottleneck tasks because their computation costs are relatively higher than other tasks. We can see that the receiving time of the rest of tasks are much larger than their computation time. A significant portion of idle time waiting for the completion of weight tasks is in the receiving time. On the other hand, we observe 23% improvement in the latency. This is because the computation time is reduced in the last two tasks with more processors assigned. T_5 and T_6 in equation (2) decrease and therefore the latency is reduced.

4 Conclusions

In this paper we presented performance results for a PRI-staggered post-Doppler STAP algorithm implementation on the Intel Paragon machine at Air Force Research Laboratory, Rome, New York. The results indicate that our approach of parallel pipelined implementation scales well both in terms of communication and computation. For the integrated pipeline system, the throughput and latency also demonstrate the linear scalability of our design. Our design and implementation not only shows tradeoffs in parallelization, processor assignment, and various overheads in inter and intra-task communication etc., but it also shows that accurate performance measurement of these systems is very important. Consideration of issues such as cache performance when data is packed and unpacked, and impact of the parallelization and processor assignment for one task on another task are crucial. This is normally not easily captured in theoretical models. In the future we plan to incorporate further optimizations including multi-threading, multiple pipelines and multiple processors on each compute node.

Acknowledgments

This work was supported by Air Force Materials Command under contract F30602-97-C-0026. We acknowledge the use of the Intel Paragon at Caltech for initial development.

References

- [Brown and Linderman, 1997] Brown, R. and Linderman, R. (1997). "Algorithm Development for an Airborne Real-Time STAP Demonstration." *IEEE National Radar Conference*.
- [Linderman and Linderman, 1997] Linderman, M. and Linderman, R. (1997). "Real-Time STAP Demonstration on an Embedded High Performance Computer." *IEEE National Radar Conference*.
- [Little and Berry, 1997] Little, M. and Berry, W. (1997). "Real-Time MultiChannel Airborne Radar Measurements." *IEEE National Radar Conference*.
- [Snir and et al., 1995] Snir, M. and et al. (1995). *MPI The Complete Reference*. The MIT Press.

Multi-Threaded Design and Implementation of Parallel Pipelined STAP on Parallel Computers with SMP Nodes

Wei-keng Liao[†], Alok Choudhary[‡], Donald Weiner[†], and Pramod Varshney[†]

[†] EECS Department
Syracuse University
Syracuse, NY 13244

[‡] ECE Department
Northwestern University
Evanston, IL 60208

Abstract

This paper presents performance results for the multi-threaded design and implementation of a parallel pipelined Space-Time Adaptive Processing (STAP) algorithm on parallel computers with Symmetrical Multiple Processor (SMP) nodes. In particular, the paper describes our approach to parallelization and multi-threaded implementation on an Intel Paragon MP system. Our goal is to determine how much more performance can be enhanced using small SMPs on each node of a large parallel computer for such an application. The paper also discusses the process of developing software for such an application on parallel computers when latency and throughput are both considered together and presents their tradeoffs. The results show that not only scalable performance was achieved for individual component tasks of STAP but linear speedups were obtained for the integrated task performance, both for latency as well as throughput.

1 Introduction

Space-time adaptive processing (STAP) is a well known technique in the area of airborne surveillance radars, which is used to detect weak target returns embedded in strong ground clutter, interference, and receiver noise. Most STAP applications consume great amounts of computational resources and are also required to operate in real time. High performance computers are becoming mainstream due to the progress made in hardware as well as software support in the last few years. They can satisfy the STAP computational requirements of real-time applications while increasing the flexibility, affordability, and scalability of radar signal processing systems. However, efficient parallelization of a STAP algorithm which has embedded in it different algorithms, is challenging and requires several optimizations.

In our previous work [3], we described the parallel

pipelined implementation of a PRI-staggered post-Doppler STAP algorithm. In this paper, we focus on the multi-threaded design and implementation on the parallel computers with SMP nodes. This STAP algorithm consists of five steps: 1) Doppler filter processing, 2) weight computation, 3) beamforming, 4) pulse compression, and 5) CFAR processing. For our implementation of this real application we designed a model of the parallel pipeline system where each pipeline is a collection of tasks and each task itself is parallelized. This parallel pipeline model was applied to the STAP algorithm with each step as a task in a pipeline. This permits us to significantly improve latency as well as throughput. Performance results presented in this paper were obtained on the Intel Paragon at the Air Force Research Laboratory (AFRL), Rome, New York.

The Intel Paragon at the AFRL is an MP system which has three processors on each compute node board. In this paper, we focus on the design of the parallel pipeline system and its implementation using multi-threading on this system. We demonstrate the performance and scalability on different numbers of compute nodes for both threaded and non-threaded implementations. The improvement of threaded implementation over non-threaded implementation is provided.

The rest of the paper is organized as follows: in Section 2, we present the parallel pipeline system model and discuss some parallelization issues. Section 3 describes the multi-threaded programming environment on the Intel Paragon MP system. Section 4 presents the implementation. Performance results and conclusions are given in Section 5 and Section 6 respectively.

2 Model of the parallel pipeline system

The system model for the type of STAP applications considered in this work is shown in Figure 1. This model is suitable for the computational characteristics found in these applications. A pipeline is a collection of tasks which are

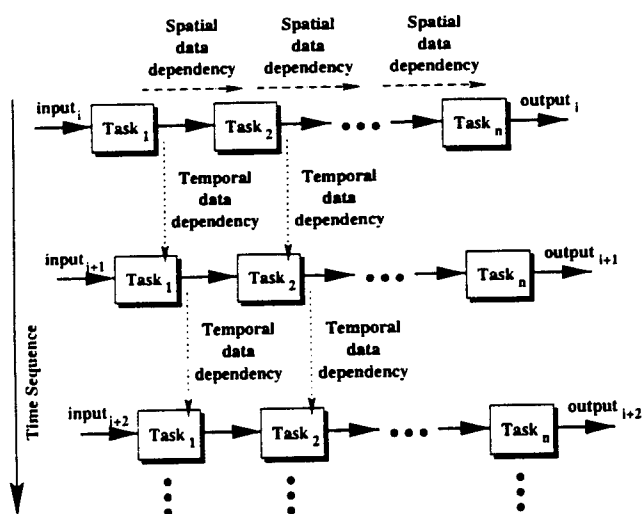


Figure 1. Model of the parallel pipeline system. The set of pipelines indicates that the same pipeline is repeated on subsequent input data sets. Each task for all input instances is executed on the same number of compute nodes.

executed sequentially. The input to the first task is obtained normally from sensors or other input devices and the inputs to the rest of the tasks in the pipeline are the outputs of their previous tasks. The set of pipelines shown in the figure indicates that the same pipeline is repeated on subsequent input data sets. Each block in a pipeline represents one parallel task, which itself is parallelized on multiple (different number of) compute nodes.

From a single task point of view, the execution flow consists of three phases: receive, compute, and send phases. In the receive and send phases, communication involves data transfer between two different groups of compute nodes. In the compute phase, work load is evenly partitioned among all compute nodes assigned in each task to achieve the maximum efficiency. For the parallel systems with SMP nodes, multi-threading technique can be employed to further improve the computation performance.

2.1 Data dependency

In such a parallel pipeline system, there exist both spatial and temporal parallelism that result in two types of data dependencies, namely, spatial data dependency and temporal data dependency [2, 5]. Spatial data dependency can be classified into inter-task data dependency and intra-task data dependency. Intra-task data dependencies arise when a set of subtasks needs to exchange intermediate results during the execution of a parallel task in a pipeline. Inter-task

data dependency is due to the transfer and reorganization of data passed onto the next parallel task in the pipeline. Temporal data dependency occurs when some form of output generated by the tasks executed on the previous data set are needed by tasks executing the current data set. We will later see that STAP has both types of data dependencies.

2.2 Compute node assignment

Optimal use of resources is particularly important in high-performance embedded applications due to limited resources and other constraints such as desired latency or throughput [4]. When several parallel tasks need to be executed in a pipelined fashion, tradeoffs exist between the assignment of processors for the maximization of overall throughput as opposed to the minimization of a single data set's response time (or latency.) The throughput requirement says that when allocating processors to tasks, it should be guaranteed that all the input data sets will be handled in a timely manner. That is, the processing rate should not fall behind the input data rate. The response time criteria, on the other hand, require minimizing the latency of computation on a particular set of data input.

3 Multi-threads on Paragon

We implemented our parallel pipeline model of the STAP algorithm on the Intel Paragon XP/S parallel computer located at AFRL. The compute partition of this machine consists of 232 MP nodes, each has three i860 processors on its compute node board. By running UNIX OSF/1 operating system, the three processors are configured with two processors as general application processors and one processor as message coprocessor which is dedicated to message passing. Multi-threaded programming environment is supported on a Paragon system and the threads are implemented as *POSIX threads* [6].

4 Design and implementation

The STAP algorithm we implemented is a PRI-staggered post-Doppler STAP algorithm [1, 7]. The design of the parallel pipelined STAP algorithm is shown in Figure 2. The parallel pipeline system consists of seven tasks. Both the weight computation and the beamforming tasks are divided into two parts, namely, "easy" and "hard" Doppler bins. The hard Doppler bins are those in which significant ground clutter is expected and the remaining bins are easy Doppler bins. The main difference between the two is the amount of data used and the amount of computation required. The input data set for the pipeline is obtained from a phased array radar and is formed in terms of a coherent processing

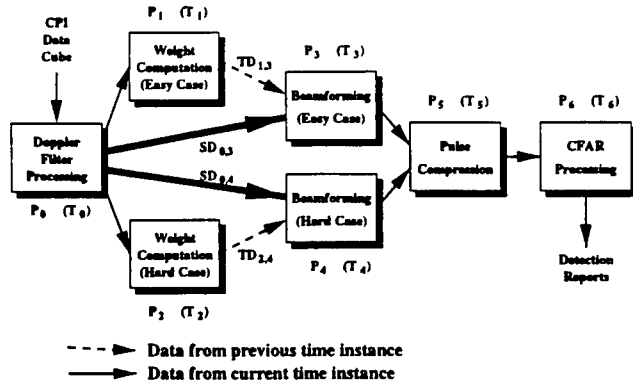


Figure 2. Implementation of parallel pipelined STAP. Arrows connecting task blocks represent data transfer between tasks.

interval (CPI). Each CPI data set is a 3-dimensional complex data cube. The output of the pipeline is a report on the detection of possible targets. Each task i , $0 \leq i < 7$, is parallelized by evenly partitioning its work load among P_i compute nodes. The execution time associated with task i is T_i . For the computation of the weight vectors for the current CPI data cube, data cubes from previous CPIs are used as input data. This introduces temporal data dependency. Temporal data dependencies are represented by arrows with dashed lines in Figure 2 where $TD_{i,j}$ represents temporal data dependency of task j on data from task i . In a similar manner, spatial data dependencies $SD_{i,j}$ can be defined and are indicated by arrows with solid lines.

Throughput and latency are two important measures for performance evaluation on a pipeline system.

$$throughput = \frac{1}{\max_{0 \leq i \leq 6} T_i}. \quad (1)$$

$$latency = T_0 + \max_{i=3,4} T_i + T_5 + T_6. \quad (2)$$

The temporal data dependency does not affect the latency because weight computation tasks use data from the previous time instance rather than the current CPI. The filtered CPI data cube sent to the beamforming task does not wait for the completion of its weight computation. This explains why equation (2) does not contain T_1 and T_2 . A detailed description of the STAP algorithm we used can be found in [1, 7].

4.1 Threads in compute phases

In the Intel Paragon MP system, two out of the three processors in one compute node are configured as general processors to run application code while the third as a message coprocessor which is dedicated to message passing.

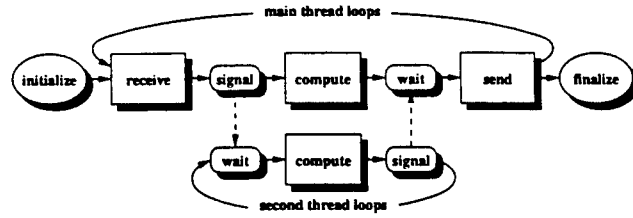


Figure 3. Implementation of two threads in the compute phase. The main thread signals the second thread to perform its computation. After completion of its computation, the second thread signals back to the main thread.

With this configuration, only compute phase for each task in our parallel pipeline system is implemented with threads. The reason for not implementing threads in communication phases is that the Paragon message-passing library is not thread-safe. Since there are only two application processors in each compute node, each compute phase in every task will have two threads implemented. Figure 3 gives the execution flows of two threads in the compute phase.

5 Performance results

The implementation of the STAP application based on our parallel pipeline system model was done on the Intel Paragon at AFRL. Each CPI complex data cube is a $512 \times 16 \times 128$ three-dimensional array. A total of 27 CPIs were generated as inputs to the parallel pipeline system.

5.1 Compute time

For each task in the STAP algorithm, parallelization was done by evenly dividing computational load across compute nodes assigned to the task. Figure 4 gives the performance results of compute phases for different tasks. For each task, we obtained linear speedups on both implementations using two threads as well as using single thread.

Assuming that the execution time of a non-threaded implementation of a task is t_1 and the execution time of its threaded implementation is t_2 , we define the threading speedup for threaded over non-threaded implementation as $s = \frac{t_1}{t_2}$. Since two processors are employed in the threaded implementation, we have $\frac{t_1}{2} \leq t_2 \leq t_1$ and, therefore, $1 \leq s \leq 2$. The threading speedups for compute phases of all tasks are also given in Figure 4. By running on two processors at the same time, the two-threaded STAP code ideally can have a threading speedup of 2. However, in most cases, the actual threading speedups do not approach this ideal value. This may be caused by the limitation of implementation of operating system, OSF/1, and the implemen-

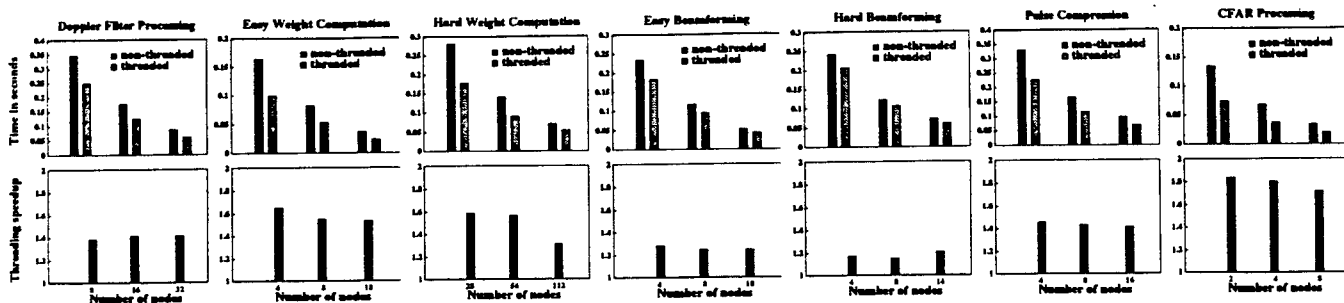


Figure 4. Performance of compute phases as a function of number of compute nodes.

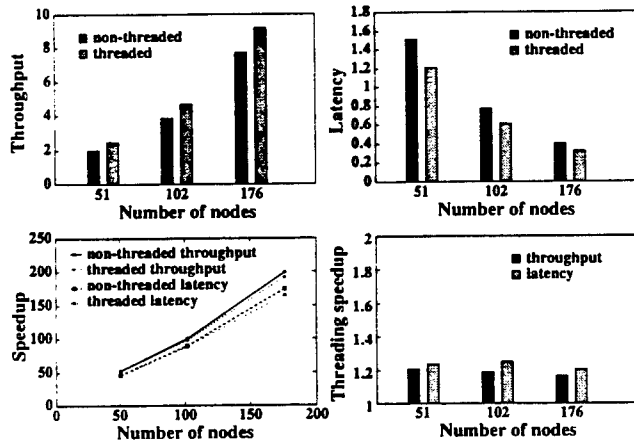


Figure 5. Integrated performance results for threaded and non-threaded implementations.

tation of linked thread-safe libraries. On an Intel Paragon MP system, scheduling of threads is handled by the operating system kernel. Users cannot have control over or get information about which processor runs which thread.

5.2 Integrated system performance evaluation

Integrated system performance evaluation refers to the evaluation of performance when all the tasks in the pipeline are considered together. Throughput (number of CPIs per second) and latency (seconds per CPI) are the two most important measures for performance evaluation on the parallel pipeline system. Figure 5 shows the speedups and threading speedups achieved by the threaded implementation for both latency and throughput for three cases of different compute node assignments with 51, 102 and 176 nodes. From these experiments, it is clear that for latency and throughput measures we obtain linear speedups for both threaded and non-threaded implementations. Given that this scale up is up to 176 compute nodes (we were limited to this number of nodes due to the size of the machine), we believe these are very good results.

5.3 Tradeoff between throughput and latency

Using an example, we illustrate how further performance improvements may (or may not) be achieved if a few additional compute nodes are available. We now take the case with 102 nodes from Figure 5 as an example and add some nodes to the pipeline to analyze its effect on the throughput and latency. Compute nodes were added to each task in increments of two nodes at a time. The resulting throughput and latency are plotted in Figure 6.

When nodes were added to the Doppler filter processing task, the throughput increased and latency reduced. From Equations (1) and (2), this improvement was obtained because the execution time, T_0 , is reduced. However, when the number of nodes added is more than 8, both throughput and latency degrade. This is because the Doppler filter processing task finishes its computation on the new CPI so fast that the actual send operations for the previous CPI have not been carried out yet. The waiting time increases Doppler filter processing task's execution time, T_0 , and therefore degrades the throughput and latency.

When compute nodes are added to easy and hard weight computation tasks, the resulting throughput and latency have no significant changes. This is because the latency does not contain the execution time of weight computations, as indicated in Equation (2). However, when extra compute nodes are added to either the beamforming or the pulse compression task, we observe that the latency is reduced. This is because the execution times T_3 , T_4 , and T_5 reduce in Equation (2). The throughput, on the other hand, is not improved because the Doppler filter processing task is the task with the maximum execution time among all tasks.

Figure 6 presents the tradeoffs between increasing the throughput and reducing the latency, when assigning nodes to the tasks in the pipeline. We observed that only the addition of nodes to the Doppler filter processing task can increase the throughput. Similarly, only beamforming and pulse compression tasks are candidates for the addition of more compute nodes to reduce the latency.

Compute node assignment can also be made in such a way that both throughput and latency are improved simul-

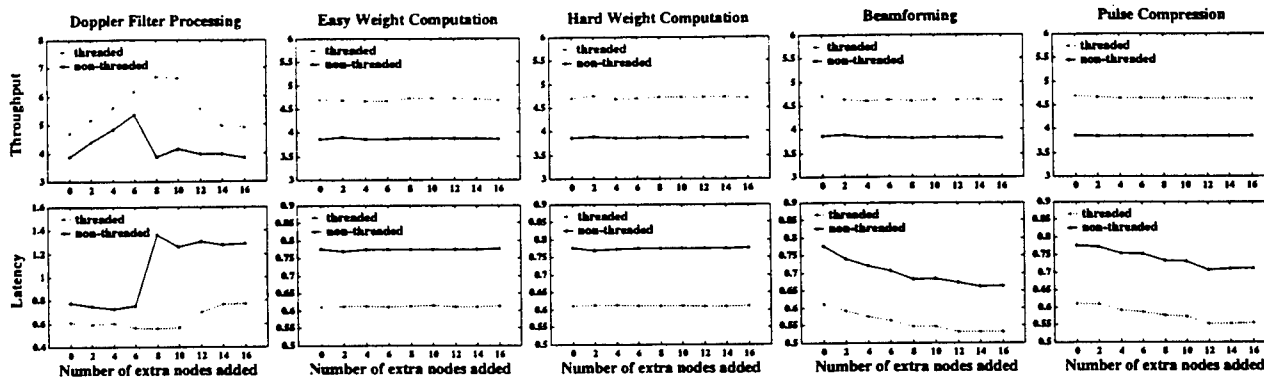


Figure 6. Throughput and latency results by adding 2 compute nodes at a time to each task.

Table 1. Performance results when 4 nodes to the Doppler processing task and 4 nodes to the pulse compression task are added to the implementation with 102 nodes.

	non-threaded		threaded	
# nodes	102	110	102	110
throughput	3.8677	4.8368	4.6916	5.6137
latency	0.7767	0.6650	0.6108	0.5458
throughput: CPIs/sec		latency: sec/CPI		

taneously. We now add 4 nodes to the Doppler filter processing task and 4 nodes to the pulse compression task. By increasing the number of compute nodes by 7.8%, the improvement in throughput is 25.1% and in latency it is 14.4% for the non-threaded implementation. Meanwhile, the threaded implementation shows 19.7% improvement in throughput and 10.6% improvement in latency. From these experimented results, we can draw the following conclusions. Extra compute nodes can be assigned to the task that has the maximum execution time among all tasks. In this way, the execution time of this task is reduced and according to Equation (1), the throughput is increased. Extra compute nodes can be added to those tasks which benefit the most, that is, the tasks with greatest reduced execution time when more nodes are assigned. The sum of these tasks can be reduced the most and therefore it minimizes the latency.

6 Conclusions

In this paper we presented performance results for a PRI-staggered post-Doppler STAP algorithm implementation on the Intel Paragon machine at Air Force Research Laboratory, Rome, New York. This Paragon machine has three processors on each compute node board. By taking advantage of the SMP architecture, a multi-threaded im-

plementation is was designed and compared to the non-threaded implementation. Performance results indicate that our approach of parallel pipelined implementation scales well both in terms of throughput and latency whether the multi-threaded technique is used or not. Our design and implementation not only shows tradeoffs in parallelization, compute node assignment, and various overheads in inter-task communication etc., but it also shows that accurate performance measurement of these systems is very important.

7 Acknowledgments

This work was supported by Air Force Materials Command under contract F30602-97-C-0026. We are grateful to Russell Brown, Mark Linderman, Richard Linderman, and Zen Pryk for their help, support, and encouragement during the course of this work.

References

- [1] R. Brown and R. Linderman. "Algorithm Development for an Airborne Real-Time STAP Demonstration,". *IEEE National Radar Conference*, 1997.
- [2] A. Choudhary. *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems*. Kluwer Academic Publisher, Boston, MA, 1990.
- [3] A. Choudhary, W. Liao, D. Weiner, P. Varshney, R. Linderman, and M. Linderman. "Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers,". *International Parallel Processing Symposium*, 1998.
- [4] A. Choudhary, B. Narahari, D. Nicol, and R. Simha. "Optimal Processor Assignment for Pipeline Computations,". *IEEE Trans. on Parallel and Distributed Systems*, Apr. 1994.
- [5] A. Choudhary and R. Ponnusamy. "Parallel Implementation and Evaluation of a Motion Estimation System Algorithm using Several Data Decomposition Strategies,". *Journal of Parallel and Distributed Computing*, Jan. 1992.
- [6] Intel Corporation. *Paragon System User's Guide*, Apr. 1996.
- [7] M. Linderman and R. Linderman. "Real-Time STAP Demonstration on an Embedded High Performance Computer,". *IEEE National Radar Conference*, 1997.

Appendix B

Papers submitted for publication

Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers

Alok Choudhary

ECE Department
Northwestern University
Evanston, IL 60208

Wei-keng Liao
Donald Weiner
Pramod Varshney

EECS Department
Syracuse University
Syracuse, NY 13244

Richard Linderman[†]
Mark Linderman[†]
Russell Brown[‡]

Information Directorate[†]
Sensors Directorate[‡]
Air Force Research Laboratory

Abstract

This paper presents performance results for the design and implementation of parallel pipelined Space-Time Adaptive Processing (STAP) algorithms on parallel computers. In particular, the paper describes the issues involved in parallelization, our approach to parallelization and performance results on an Intel Paragon. The paper also discusses the process of developing software for such an application on parallel computers when latency and throughput are both considered together and presents tradeoffs considered with respect to inter and intra-task communication and data redistribution. The results show that not only scalable performance was achieved for individual component tasks of STAP but linear speedups were obtained for the integrated task performance, both for latency as well as throughput. Results are presented for up to 236 compute nodes (limited by the machine size available to us). Another interesting observation made from the implementation results is that performance improvement due to the assignment of additional processors to one task can improve the performance of other tasks without any increase in the number of processors assigned to them. Normally, this cannot be predicted by theoretical analysis.

1 Introduction

Space-time adaptive processing (STAP) is a well known technique in the area of airborne surveillance radars used to detect weak target returns embedded in strong ground clutter, interference, and receiver noise. STAP is a 2-dimensional adaptive filtering algorithm that attenuates unwanted signals by placing nulls in their directions of arrival and Doppler frequencies. Most STAP applications are computationally intensive and must operate in real time. High performance computers are becoming mainstream due to the progress made in hardware as well as software support in the last few years. They can satisfy the STAP computational requirements of real-time applications while increasing the flexibility, affordability, and scalability of radar signal processing systems. However, efficient parallelization of a STAP algorithm which has embedded in it different processing steps is challenging and is the subject of this paper.

This paper describes our innovative parallel pipelined implementation of a Pulse Repetition Interval (PRI)-staggered post-Doppler STAP algorithm on the Intel Paragon at the Air Force Research Laboratory (AFRL), Rome, New York. For a detailed description of the STAP algorithm implemented in this work, the reader is referred to [1, 2]. AFRL successfully installed their implementation of the STAP algorithm onboard an airborne platform and performed four flight experiments in May and June 1996 [3]. These experiments were performed as part of the Real-Time Multi-Channel Airborne Radar Measurements (RTMCARM) program. The RTMCARM system block diagram is shown in Figure 1. In that real-time demonstration, live data from a phased array radar was processed by the onboard Intel Paragon and results showed that high performance computers can deliver a significant performance gain. However, this implementation used compute nodes of the machine only as independent resources in a round robin fashion to run different instances of STAP (rather than speeding up each instance of STAP.) Using this approach, the throughput may be improved, but the latency is limited by what can be achieved using one compute node.

Parallel computers, organized with a large set (several hundreds) of processors linked by a specialized high speed interconnection network, offer an attractive solution to many computationally intensive applications, such as image processing, simulation of particle reactions, and so forth. Parallel processing splits an application problem into several subproblems which are solved on

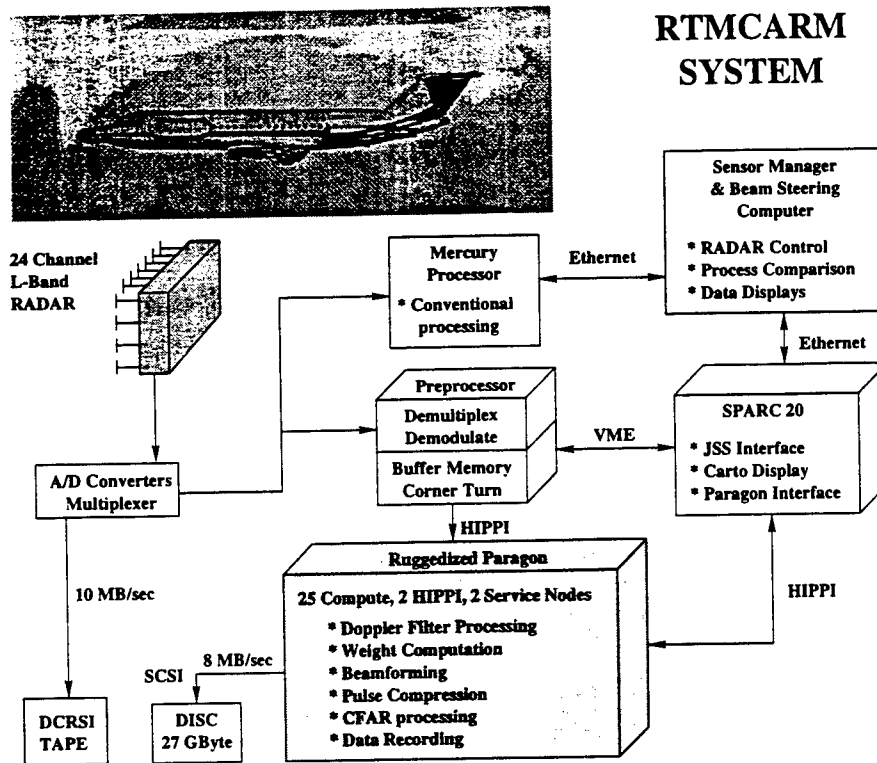


Figure 1. RTMCARM system block diagram.

multiple processors simultaneously. To learn more about parallel computing, the reader is referred to [4, 5, 6, 7, 8]. For our parallel implementation of this real application we have designed a model of the parallel pipeline system where each pipeline is a collection of tasks and each task itself is parallelized. This parallel pipeline model was applied to the STAP algorithm with each step as a task in a pipeline. This permits us to significantly improve latency as well as throughput.

This paper describes parallelization process and performance results. In addition, design considerations for portability, task mapping, parallel data redistribution, parallel pipelining as well as system-level and task-level performance measurement are discussed. Finally, the performance and scalability of the implementation for a large number of processors is demonstrated. Performance results are presented for the Intel Paragon at AFRL.

The paper is organized as follows. In Section 2 we discuss the related work. An overview of the implemented algorithm is given in Section 3. In Section 4, we present the parallel pipeline system model and discuss some parallelization issues and approaches for implementation of STAP algorithms. Section 5 presents specific details of STAP implementation. Performance results and

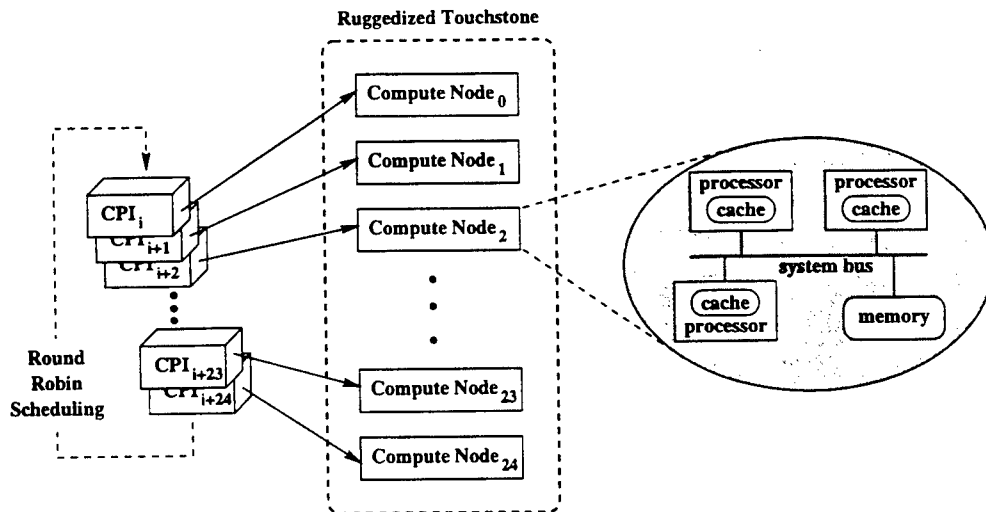


Figure 2. Implementation of the ruggedized version of Intel Paragon system in RTMCARM experiments.

conclusions are presented in Section 7 and Section 8, respectively.

2 Related Work

The RTMCARM experiments were performed using a BAC 1-11 aircraft. The radar was a phased array L-Band radar with 32 elements organized into two rows of 16 each. Only the data from the upper 16 elements were processed with STAP. This data was derived from a 1.25 MHz intermediate frequency (IF) signal that was 4:1 oversampled at 5 MHz. The number representation at IF was 14 bits, 2's complement and was converted to 16 bit baseband real and imaginary numbers. Special interface boards were used to digitally demodulate IF signals to baseband. The signal data formed a raw 3-dimensional data cube called coherent processing interval (CPI) data cube comprised of 128 pulses, 512 range gates (32.8 miles), and 16 channels. These special interface boards were also used to corner turn the data cube so that CPI is unit stride along pulses. It speeds the subsequent Doppler processing on the High Performance Computing (HPC) systems. Live CPI data from a phased-array radar were processed by a ruggedized version of the Paragon computer.

The ruggedized version of Intel Paragon system consists of 25 compute nodes running the SUN-MOS operating system. Figure 2 depicts the system implementation. Each compute node has three

i860 processors accessing the common memory of size 64M bytes as a shared resource. The CPI data sets were sent to the 25 compute nodes in a round robin manner and all three processors worked on each CPI data set as a shared-memory machine. The system processed up to 10 CPIs per seconds (throughput) and achieved a latency of 2.35 seconds per CPI. This implementation used compute nodes of the machine as independent resources to run different instances of CPI data sets. No communication among compute nodes was needed. This approach can achieve desired throughput by using as many nodes as needed, but the latency is limited by what can be achieved using the three processors in one compute node. More information on the overall system configuration and performance results can be found in [1, 3].

Other related work such as [9, 10, 11, 12] parallelized high-order post-Doppler STAP algorithms by partitioning the computational workload among all processors allocated for the applications. In [9, 10], they focused on the design of parallel versions of subroutines for FFT and QR decomposition. In [11, 12], the implementations optimized the data redistribution between processing steps in the STAP algorithms while using sequential versions of FFT and QR decomposition subroutines. A multi-stage approach was employed in [13] which was an extension of [11, 12]. A beam space post-Doppler STAP was divided into three stages and each stage was parallelized on a group of processors. A technique called replication of pipeline stages was used to replicate the computational intensive stages such that different data instance is run on a different replicated stage. Their effort focused on increasing the throughput while keeping the latency fixed. For other related work, the reader is referred to [14, 15, 16].

3 Algorithm Overview

The adaptive algorithm, which cancels Doppler shifted clutter returns as seen by the airborne radar system, is based on a least squares solution to the weight vector problem. This approach has traditionally yielded high clutter rejection, but suffers from severe distortions in the adapted main beam pattern and resulting loss of gain on the target. Our approach introduces a set of constraint equations into the least squares problem which can be weighted proportionally to preserve main beam shape. The algorithm is structured so that multiple receive beams may be formed without changing the matrix of training data. Thus, the adaptive problem can be solved once for all beams

which lie within the transmit illumination region. The airborne radar system was programmed to transmit five beams, each 25 degrees in width, spaced 20 degrees apart. Within each transmit beam, six receive beams were formed by the processor.

The algorithm consists of the following steps:

1. Doppler filter processing,
2. Weight computation,
3. Beamforming,
4. Pulse compression, and
5. CFAR processing.

Doppler filtering is performed on each receive channel using weighted Fast Fourier Transforms (FFT's). The analog portion of the receiver compensates the received clutter frequency to center the clutter frequency at zero regardless of the transmit beam position. This simplifies indexing of Doppler bins for classification as "easy" or "hard" depending on their proximity to mainbeam clutter returns. For the hard cases, Doppler processing is performed on two 125-pulse windows of data separated by three pulses (a STAP technique known as "PRI-stagger"). Both sets of Doppler processed data are adaptively weighted in the beamforming process for improved clutter rejection. In the easy case, only a single Doppler spectrum is computed. This simpler technique has been termed Post Doppler Adaptive Beamforming and is quite effective at a fraction of the computational cost when the Doppler bin is well separated from mainbeam clutter. In these situations, an angular null placed in the direction of the competing ground clutter provides excellent rejection. Selectable window functions are applied to the data prior to the Doppler FFT's to control sidelobe levels. The selection of a window is a key parameter in that it impacts the leakage of clutter returns across Doppler bins, traded off against the width of the clutter passband.

An efficient method of beamforming using recursive weight updates is made possible by a block update form of the QR decomposition algorithm. This is especially significant in the hard Doppler regions, which are computed using separate weights for six consecutive range intervals. The recursive algorithm requires substantially less training data (sample support) for accurate weight computation, as well as providing improved efficiency. Since the hard regions have one sixth the

range extent from which to draw data, this approach dealt with the paucity of data by using past looks at the same azimuth, exponentially forgotten, as independent, identically distributed estimates of the clutter to be cancelled. This assumes a reasonable revisit time for each azimuth beam position. During the flight experiments, the five 25 degree transmit beam positions were revisited at a 1-2 Hz rate (5-10 CPIs per second.)

The training data for the easy Doppler regions was selected using a more traditional approach. Here, the entire range extent was available for sample support, so the entire training set was drawn from three preceding CPIs for application to the next CPI in this azimuth beam position. In this case, a regular (non-recursive) QR decomposition is performed on the training data, followed by block update to add in the beam shape constraints.

Pulse compression is a compute intensive task, especially if applied to each receive channel independently. In general, this approach is required for adaptive algorithms which compute different weight sets as a function of radar range. Our algorithm, however, with its mainbeam constraint, preserves phase across range. In fact, the phase of the solution is independent of the clutter nulling equations, and appears only in the constraint equations. The adapted target phase is preserved across range, even though the clutter and adaptive weights may vary with range. Thus, pulse compression may be performed on the beamformed output of the receive channels providing a substantial savings in computations.

In the sections to follow, we present the process of parallelization and software design considerations including those for portability, task mapping, parallel data redistribution, parallel pipelining and issues involved in measuring performance in implementations when not only the performance of individual tasks is important, but overall performance of the integrated system is critical. We demonstrate the performance and scalability for a large number of processors.

4 Model of the Parallel Pipelined System

The system model for the type of STAP applications considered in this work is shown in Figure 3. A pipeline is a collection of tasks which are executed sequentially. The input to the first task is obtained normally from sensors or other input devices with the inputs to the remaining tasks coming from outputs of previous tasks. The set of pipelines shown in the figure indicates that

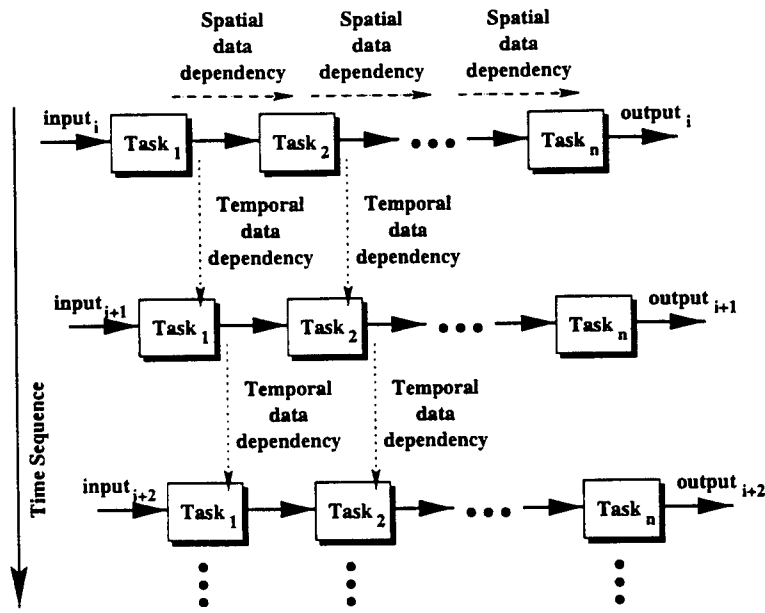


Figure 3. Model of the parallel pipeline system. (Note that $Task_i$ for all input instances is executed on the same number of processors, but that the number of processors may differ from one task to another.)

the same pipeline is repeated on subsequent input data sets. Each block in a pipeline represents one task, that is parallelized on multiple (different number of) processors. That is, each task is decomposed into subtasks to be performed in parallel. Therefore, each pipeline is a collection of parallel tasks.

In such a system, there exist both spatial and temporal parallelism that result in two types of data dependencies and flows, namely, spatial data dependency and temporal data dependency [17, 18, 19]. Spatial data dependency can be classified into inter-task data dependency and intra-task data dependency. Intra-task data dependencies arise when a set of subtasks needs to exchange intermediate results during the execution of a parallel task in a pipeline. Inter-task data dependency is due to the transfer and reorganization of data passed onto the next parallel task in the pipeline. Inter-task communication can be communication from the subtasks of the current task to the subtasks of the next task, or collection and reorganization of output data of the current task and then redistribution of the data to the next task. The choice depends on the underlying architecture, mapping of algorithms and input-output relationship between consecutive tasks. Temporal data

dependency occurs when some form of output generated by the tasks executed on the previous data set are needed by tasks executing the current data set. STAP is an interesting parallelization problem because it exhibits both types of data dependency.

4.1 Parallelization Issues and Approaches

A STAP algorithm involves multiple algorithms (or processing steps), each of which performs particular functions, to be executed in a pipelined fashion. Multiple pipelines need to be executed in a staggered manner to satisfy the throughput requirements. Each task needs to be parallelized for the required performance, which, in turn, requires addressing the issue of data distribution on the subset of processors on which a task is parallelized to obtain good efficiency and incur minimal communication overhead. Given that each task is parallelized, data flow among multiple processors of two or more tasks is required and, therefore, communication scheduling techniques become critical.

4.1.1 Inter-task Data Redistribution

In an integrated system, data redistribution is required to feed data from one parallel task to another, because the way data is distributed in one task may not be the most appropriate distribution for the next task for algorithmic or efficiency reasons. For example, the FFTs in the Doppler filter processing task perform optimally when the data is unit-stride in pulse, while the next stage, beamforming, performs optimally when the data is unit stride in channel. To ensure efficiency and continuity of memory access, data reorganization and redistribution are required in the inter-task communication phase. Data redistribution also allows concentration of communication at the beginning and the end of each task.

We have developed runtime functions and strategies that perform efficient data redistribution [20]. These techniques reduce the communication time by minimizing contention on the communication links as well as by minimizing the overhead of processing for redistribution (which adds to the latency of sending messages). We take advantage of lessons learned from these techniques to implement the parallel pipelined STAP application.

4.1.2 Task Scheduling and Processor Assignment

An important factor in the performance of a parallel system is how the computational load is mapped onto the processors in the system. Ideally, to achieve maximum parallelism, the load must be evenly distributed across the processors. The problem of statically mapping the workload of a parallel algorithm to processors in a distributed memory system has been studied under different problem models, such as [21, 22]. The mapping policies are adequate when an application consists of a single task, and the computational load can be determined statically. These static mapping policies do not model applications consisting of a sequence of tasks (algorithms) where the output of one task becomes the input to the next task in the sequence.

Optimal use of resources is particularly important in high-performance embedded applications due to limited resources and other constraints such as desired latency or throughput [23]. When several parallel tasks need to be executed in a pipelined fashion, tradeoffs exist between assigning processors to maximize the overall throughput and assigning processors to minimize a single data set's response time (or latency.) The throughput requirement says that when allocating processors to tasks, it should be guaranteed that all the input data sets will be handled in a timely manner. That is, the processing rate should not fall behind the input data rate. The response time criteria, on the other hand, require minimizing the latency of computation on a particular set of data input.

To reduce the latency, each parallel task must be allocated more processors to reduce its execution time, and consequently, the overall execution time of the integrated system. But it is well known that the efficiency of parallel programs usually decreases as the number of processors is increased. Therefore, the gains in this approach may be incremental. On the other hand, throughput can be increased by increasing the latency of individual tasks by assigning them fewer processors and, therefore, increasing efficiency, but at the same time having multiple streams active concurrently in a staggered manner to satisfy the input-data rate requirements. We next present these tradeoffs and discuss various implementation issues.

5 Design and Implementation

The design of the parallel pipelined STAP algorithm is shown in Figure 4. The parallel pipeline system consists of seven basic tasks. We refer to the parallel pipeline as simply a pipeline in the rest

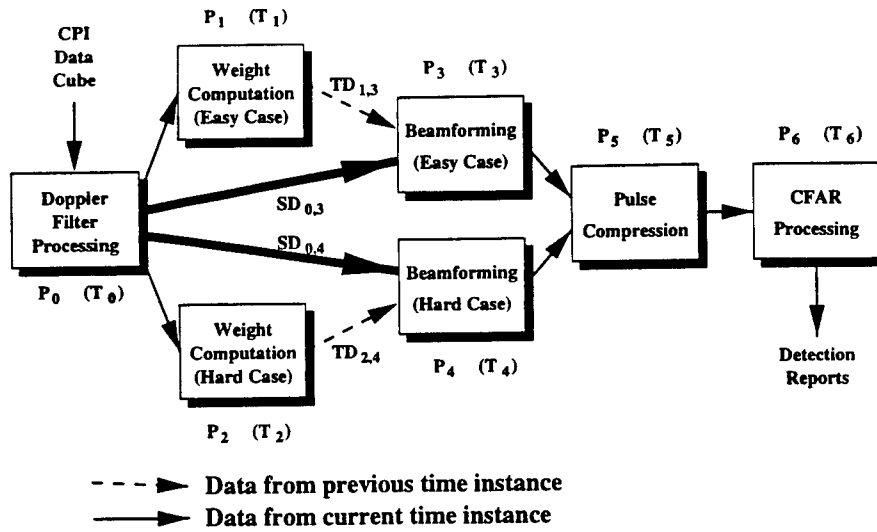


Figure 4. Implementation of parallel pipelined STAP. Arrows connecting task blocks represent data transfer between tasks.

of this paper. The input data set for the pipeline is obtained from a phased array radar and is formed in terms of a coherent processing interval (CPI). Each CPI data set is a 3-dimensional complex data cube comprised of K range cells, J channels, and N pulses. The output of the pipeline is a report on the detection of possible targets. The arrows shown in Figure 4 indicate data transfer between tasks. Although a single arrow is shown, note that each represents multiple processors in one task communicating with multiple processors in another task. Each task i is parallelized by evenly partitioning its work load among P_i processors. The execution time associated with task i , T_i , consists of the time to receive data from the previous task, computation time, and time to send results to the next task.

The calculation of weights is the most computationally intensive part of the STAP algorithm. For the computation of the weight vectors for the current CPI data cube, data cubes from previous CPIs are used as input data. This introduces temporal data dependency. For example, suppose that a set of CPI data cubes entering the pipeline sequentially are denoted by CPI_i , $i = 0, 1, \dots$. At any time instance i , the Doppler filtering task is processing CPI_i and beamforming task is processing CPI_{i-1} . In the meanwhile, the weight computation task is using past CPIs in the same azimuthal direction to calculate the weight vectors for CPI_i as described below. The computed weight vectors will be applied to CPI_i in the beamforming task at next time instance $i + 1$. Thus,

temporal data dependencies exist and are represented by arrows with dashed lines, $TD_{1,3}$ and $TD_{2,4}$, in Figure 4 where $TD_{i,j}$ represents temporal data dependency of task j on data from task i . In a similar manner, spatial data dependencies $SD_{i,j}$ can be defined and are indicated in Figure 4 by arrows with solid lines.

Throughput and latency are two important measures for performance evaluation on a pipeline system. The throughput of our pipeline system is the inverse of the maximum execution time among all tasks, i.e.,

$$throughput = \frac{1}{\max_{0 \leq i < 7} T_i}. \quad (1)$$

To maximize the throughput, the maximum value of T_i should be minimized. In other words, no task should have an extremely large execution time. With a limited number of processors, the processor assignment to different tasks must be made in such a way that the execution time of the task with highest computation time is reduced.

The latency of this pipeline system is the time between the arrival of the CPI data cube at the system input and the time at which the detection report is available at the system output. Therefore, the latency for processing one CPI is the sum of the execution times of all the tasks except weight computation tasks, i.e.,

$$latency = T_0 + \max(T_3, T_4) + T_5 + T_6. \quad (2)$$

Equation (2) does not contain T_1 and T_2 . The temporal data dependency does not affect the latency because weight computation tasks use data from the previous instance of CPI data rather than the current CPI. The filtered CPI data cube sent to the beamforming tasks do not wait for the completion of its weight computation but rather for the completion of the weight computation of the previous CPI. For example, when the Doppler filter processing task is processing CPI_i , the weight computation tasks use the filtered CPI data, CPI_{i-1} , to calculate the weight vectors for CPI_i . At the same time, the beamforming tasks are working on CPI_{i-1} using the data received from the Doppler filter processing and weight computation tasks. The beamforming tasks do not wait for the completion of the weight computation task when processing CPI_{i-1} data. The overall system latency can be reduced by reducing the execution times of the parallel tasks, e.g., T_0 , T_3 , T_4 , T_5 , and T_6 in our system.

Next, we briefly describe each task and its parallel implementation. A detailed description of

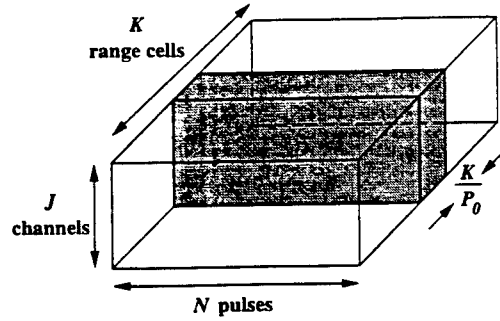


Figure 5. Partitioning strategy for Doppler filter processing task. The CPI data cube is partitioned among P_0 processors across dimension K .

the STAP algorithm we used can be found in [1, 2].

5.1 Doppler Filter Processing

The input to the Doppler filter processing task is one CPI complex data cube received from a phased array radar. The computation in this task involves performing range correction for each range cell and the application of a windowing function (e.g. Hanning or Hamming) followed by a N -point FFT for every range cell and channel. The output of the Doppler filter processing task is a 3-dimensional complex data cube of size $K \times 2J \times N$ which is referred to as staggered CPI data. In Figure 4, we can see that this output is sent to the weight computation task as well as to the beamforming task.

Both the weight computation and the beamforming tasks are divided into easy and hard parts. These two parts use different portions of staggered CPI data and the associated amounts of computation are also different. The easy weight computation task uses range samples only from the first half of the staggered CPI data while the hard weight computation task uses range samples from the entire staggered CPI data. On the other hand, easy and hard beamforming tasks use all range cells rather than some of them. Therefore, the size of data to be transferred to weight computation tasks is different from the size of data to be sent to beamforming tasks. In Figure 4, thicker arrows connected from Doppler filter processing task to beamforming tasks indicates that the amount of data sent to the beamforming tasks is more than the amount of data sent to the weight tasks.

The basic parallelization technique employed in the Doppler filtering processing task is to par-

tion the CPI data cube across the range cells, that is, if P_0 processors are allocated to this task, then each processor is responsible for $\frac{K}{P_0}$ range cells. The reason for partitioning the CPI data cube along dimension K is that it maintains an efficient accessing mechanism for contiguous memory space. A total of $K \cdot 2J$ N -point FFTs are performed and the best performance is achieved when every N -point FFT accesses its N data sets from a contiguous memory space. Figure 5 illustrates the parallelization of this step. The inter-task communication from the Doppler filter processing task to weight computation tasks is explained in Figure 6(b). Since only subsets of range cells are needed in weight computation tasks, data collection has to be performed on the output data before passing it to the next tasks. Data collection is performed to avoid sending redundant data and hence reduces the communication costs.

5.2 Weight Computation

The second step in this pipeline is the computation of weights that will be applied to the next CPI. This computation for N pulses is divided into two parts, namely, "easy" and "hard" Doppler bins, as shown in Figure 6(a). The hard Doppler bins (pulses), N_{hard} , are those in which significant ground clutter is expected. The remaining bins are easy Doppler bins, N_{easy} . The main difference between the two is the amount of data used and the amount of computation required. Not all range cells in the staggered CPI are used in weight calculation and different subsets of range samples are used in easy Doppler bins and hard Doppler bins.

To gather range samples for easy Doppler bins to calculate the weight vectors for the current CPI, data is drawn from three preceding CPIs by evenly spacing out over the first one third of K range cells of each of the three CPIs. The easy weight computation task involves N_{easy} QR factorizations, block updates, and back substitutions. In the easy weight calculation, only range samples in the first half of the staggered CPI data are used while hard weight computation employs range samples from the entire staggered CPI. Furthermore, range extent for hard Doppler bins is split into six independent segments to further improve clutter cancelation. To calculate weight vectors for the current CPI, range samples used in hard Doppler bins are taken from the immediately preceding staggered CPI combined with older, exponentially forgotten, data from CPIs in the same direction. This is done for each of the six range segments. The hard weight computation task

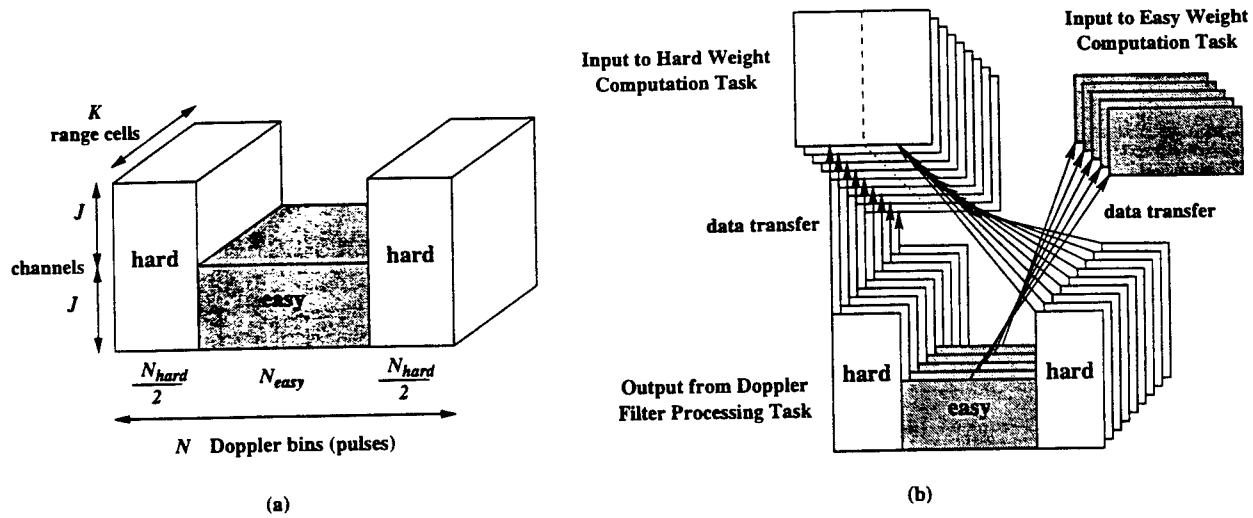


Figure 6. (a) Staggered CPI data partitioned into easy and hard weight computation tasks. (b) Parallel inter-task communication from Doppler filter processing task to easy and hard weight computation tasks requires different sets of range samples. Data collection needs to be performed before the communication. This can be viewed as irregular data redistribution.

involves $6N_{hard}$ recursive QR updates, block updates, and back substitutions. The easy and hard weight computation tasks process sets of 2-dimensional matrices of different sizes.

Temporal data dependency exists in the weight computation task because both easy and hard Doppler bins use data from previous CPIs to compute the weights for the current CPI. The outputs of this step, the weight vectors, are two 3-dimensional complex data cubes of size $N_{easy} \times J \times M$ and $N_{hard} \times 2J \times M$ for easy and hard weight computation tasks, respectively, where M is the number of receive beams. These two weight vectors are to be applied to the current CPI in the beamforming task. Because of the different sizes of easy and hard weight vectors, the beamforming task is also divided into easy and hard parts to handle different amounts of computation.

Given the uneven nature of weight computations, different sets of processors are allocated to the easy and hard tasks. In Figure 4, P_1 processors are allocated to easy weight computation and P_2 processors to hard weight computation. Since weight vectors are computed for each pulse (Doppler bin), the parallelization in this step involves partitioning of data along dimension N , that is, each processor in easy weight computation task is responsible for $\frac{N_{easy}}{P_1}$ pulses while each processor in hard weight computation task is responsible for $\frac{N_{hard}}{P_2}$ pulses, as shown in Figure 7.

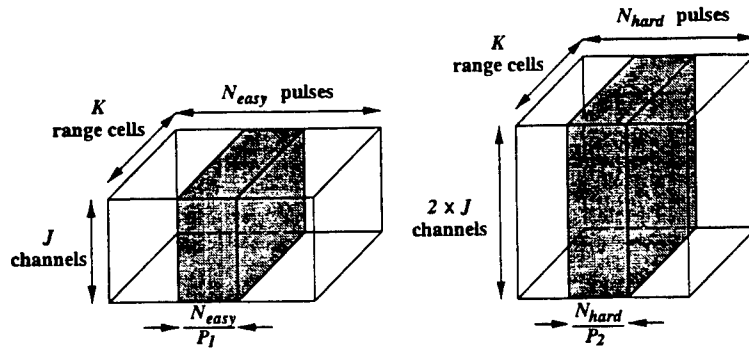


Figure 7. Partitioning strategy for easy and hard weight computation tasks. Data cube is partitioned across dimension N .

Notice that Doppler filter processing and weight computation tasks employ different data partitioning strategies (along different dimensions.) Due to different partitioning strategies, an all-to-all personalized communication scheme is required for data redistribution from Doppler filter processing task to the weight computation task. That is, each of the P_1 and P_2 processors needs to communicate with all P_0 processors allocated to the Doppler filter processing task to receive CPI data. Since only subsets of Doppler filter processing task's output are used in the weight computation task, data collection is performed before inter-task communication. Although data collection reduces inter-task communication cost, it also involves data copying from non-contiguous memory space to contiguous buffers. Sometimes the cost of data collection may become extremely large due to hardware limitations (e.g. high cache miss ratio.) When sending data to the beamforming task, the weight vectors have already been partitioned along dimension N which is the same as the data partitioning strategy for the beamforming task. Therefore, no data collection is needed when transferring data to the beamforming task.

5.3 Beamforming

The third step in this pipeline (which is actually the second step for the current CPI because the result of the weight task is only used in the subsequent time step) is beamforming. The inputs of this task are received from both Doppler filter processing and weight computation tasks, as shown in Figure 4. The easy weight vector received from easy weight computation task is applied

to the easy Doppler bins of the received CPI data while the hard weight vector is applied to hard Doppler bins. The application of weights to CPI data requires matrix-matrix multiplications on two received data sets. Due to different matrix sizes for multiplications in easy and hard beamforming tasks, uneven computational load results. The beamforming task is also divided into easy and hard parts for parallelization purposes. This is because the easy and hard beamforming tasks require different amounts and portions of CPI data, and involve different computational loads. The inputs for the easy beamforming task are two 3-dimensional complex data cubes. One data cube which is received from the easy weight computation task is of size $N_{easy} \times M \times J$. The other is from Doppler filter processing task and its size is $N_{easy} \times J \times K$. A total of N_{easy} matrix-matrix multiplications are performed where each multiplication involves two matrices of size $M \times J$ and $J \times K$, respectively. The hard beamforming task also has two input data cubes which are received from Doppler filter processing and hard weight computation tasks. The data cube of size $6N_{hard} \times M \times 2J$ is received from hard weight computation task and the Doppler filtered CPI data cube is of size $N_{hard} \times 2J \times K$. Since range cells are divided into 6 range segments, there are a total of $6N_{hard}$ matrix-matrix multiplications in hard beamforming. The results of the beamforming task are two 3-dimensional complex data cubes of size $N_{easy} \times M \times K$ and $N_{hard} \times M \times K$ corresponding to easy and hard parts respectively.

In a manner similar to the weight computation task, parallelization in this step also involves partitioning of data across the N dimension (Doppler bins.) Different sets of processors are allocated to easy and hard beamforming tasks. Since the cost of matrix multiplications can be determined accurately, the computations are equally divided among the allocated processors for this task. As seen from Figure 4, this task requires data to be communicated from the first as well as the second task. Because data is partitioned along different dimensions, an all-to-all personalized communication is required for data redistribution between Doppler filter processing and beamforming tasks. The output of the Doppler filter processing task is a data cube of size $K \times 2J \times N$ which is redistributed to the beamforming task after data reorganization in the order of $N \times K \times 2J$. Data reorganization has to be done before the inter-task communication between the two tasks takes place, as shown in Figure 8.

Data reorganization involves data copying from non-contiguous memory space and its cost may become extremely large due to cache misses. For example, two Doppler bins in the same range cell

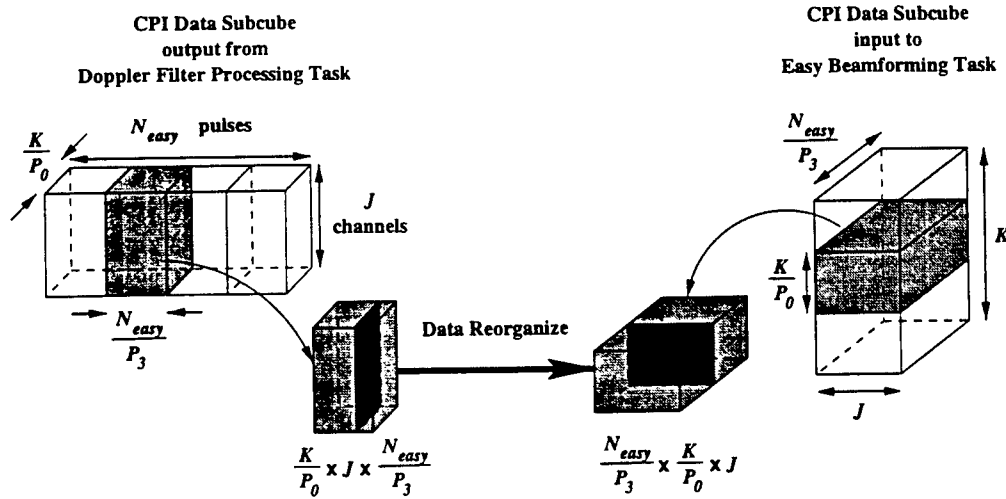


Figure 8. Data redistribution from Doppler filter processing task to easy beamforming task. CPI data subcube of size $\frac{K}{P_0} \times J \times \frac{N_{easy}}{P_3}$ is reorganized to subcube of size $\frac{N_{easy}}{P_3} \times \frac{K}{P_0} \times J$ before sending from one processor in Doppler filter processing task to another in easy beamforming task.

and the same channel are stored in contiguous memory space. After data reorganization, they are $\frac{K}{P_0} \cdot J$ element distance apart. Therefore, if P_0 is small and the size of CPI data subcube partitioned in each processor is large then it is quite likely that expensive data reorganization will be needed which becomes a major part of communication overhead. The algorithms which perform data collection and reorganization are crucial to exploit the available parallelism. Note that receiving data from weight computation tasks does not involve data reorganization or data collection because they have the same partitioning strategy (along dimension N .)

5.4 Pulse Compression

The input to the pulse compression task is a 3-dimensional complex data cube of size $N \times M \times K$, as shown in Figure 9. This data cube consists of two subcubes of size $N_{easy} \times M \times K$ and $N_{hard} \times M \times K$ which are received from easy and hard beamforming tasks respectively. Pulse compression involves convolution of the received signal with a replica of the transmit pulse waveform. This is accomplished by first performing K -point FFTs on the two inputs, point-wise multiplication of the intermediate result and then computing the inverse FFT. The output of this step is a 3-dimensional

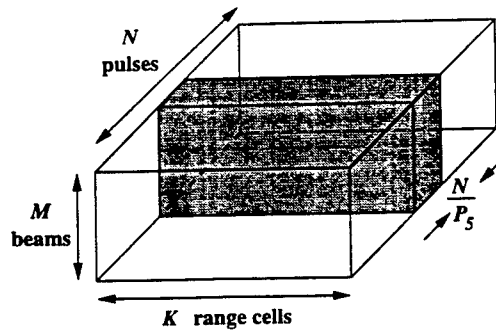


Figure 9. Partitioning strategy for pulse compression task. Data cube is partitioned across dimension N into P_s processors.

real data cube of size $N \times M \times K$. The parallelization of this step is straightforward and involves the partitioning of data cube across the N dimension. Each of the FFTs could be performed on an individual processor and hence each processor in this task gets an equal amount of computation. Partitioning along the N dimension also results in an efficient accessing mechanism for contiguous memory space when running FFTs. Since both beamforming and pulse compression tasks use the same data partitioning strategy (along dimension N), no data collection or reorganization is needed prior to communication between these two tasks. After pulse compression, the square of the magnitude of the complex data is computed to move to the real power domain. This cuts data set size in half and eliminates the computation of the square root.

5.5 CFAR Processing

The input to this task is an $N \times M \times K$ real data cube received from the pulse compression task. The sliding window constant false alarm rate (CFAR) processing compares the value of a test cell at a given range to the average of a set of reference cells around it times a probability of false alarm factor. This step involves summing up a number of range cells on each side of the cell under test, multiplying the sum by a constant, and comparing the product to the value of the cell under test. The output of this task, which appears at the pipeline output, is a list of targets at specified ranges, Doppler frequencies, and look directions. The parallelization strategy for this step is the same as for the pulse compression task. Both tasks partition data cube along the N dimension. Also, no data collection or reorganization is needed in pulse compression task before sending data to this

n : number of CPIs
inBuf[2] : input data buffer
outBuf[2] : output data buffer

```
1  for  $i \leftarrow 0$  to  $n - 1$ 
2     $prev \leftarrow (i - 1) \bmod 2$ 
3     $cur \leftarrow i \bmod 2$ 
4     $next \leftarrow (i + 1) \bmod 2$ 
5     $t_0 \leftarrow$  read timer
6    post async receives for inBuf[ $next$ ]
7    wait for completion of previous receives for inBuf[ $cur$ ]
8    data unpacking on inBuf[ $cur$ ]
9     $t_1 \leftarrow$  read timer
10   computation on inBuf[ $cur$ ] and result in outBuf[ $cur$ ]
11    $t_2 \leftarrow$  read timer
12   data packing for outgoing message on outBuf[ $cur$ ]
13   post async sends for outBuf[ $cur$ ] to next task
14   wait for completion of sends for outBuf[ $prev$ ]
15    $t_3 \leftarrow$  read timer
```

Figure 10. Implementation of timing computation and communication for each task. A double buffering strategy is used to overlap the communication with the computation. Receive time = $t_1 - t_0$, compute time = $t_2 - t_1$, and send time = $t_3 - t_2$.

task.

6 Software Development and System Platform

All the parallel program development and their integration was performed using ANSI C language and message passing interface (MPI) [24]. This permits easy portability across various platforms which support C language and MPI. Since MPI is becoming a de facto standard for

high-performance systems, we believe the software is portable.

The implementation of the STAP application based on our parallel pipeline system model has been done on the Intel Paragon at the Air Force Research Laboratory, Rome, New York. This machine contains 321 compute nodes interconnected in a two-dimensional mesh. The Paragon runs Intel's standard Open Software Foundation (OSF) UNIX operating system. Each compute node consists of three i860 RISC processors which are connected by a system bus and share a 64M byte memory. The speed of an i860 RISC processor is 40 MHz and its peak performance is 100M floating point operations per second. The interconnection network has a message startup time of 35.3 μ sec and a data transfer time of 6.53 nsec/byte for point-to-point communication.

In our implementation, a double buffering strategy was used both in receive and send phases. During the execution loops, this strategy employs two buffers alternatively such that one buffer can be processed during the communication phase while the other buffer is processed during the compute phase. Together with the double buffering implementation, asynchronous send and receive calls were employed in order to maximize the overlap of communication and computation. Asynchronous communication means that the program executing the send/receive does not wait until the send/receive is complete. This type of communication is also referred to as non-blocking communication. The other option is synchronous communication which blocks the send/receive operation till the message has been sent/received. The general execution flow and the approach to measure the timing for each part of computation and communication is given in Figure 10. We used MPI timer, `MPI_Wtime()`, because this function is portable with high resolution.

7 Performance Results

We specified the parameters that were used in our experiments as follows:

- range cells (K) = 512,
- channels (J) = 16,
- pulses (N) = 128,
- receive beams (M) = 6,

Table 1. The number of floating point operations for the PRI-staggered post Doppler STAP algorithm to process one CPI data.

Task	number of floating point operations
Doppler filter processing	79,691,776
hard weight computation	197,038,464
easy weight computation	13,851,792
easy beamforming	28,311,552
hard beamforming	44,040,192
pulse compression	38,928,384
CFAR processing	1,690,368
Total	403,552,528

- easy Doppler bins (N_{easy}) = 72, and
- hard Doppler bins (N_{hard}) = 56.

Given these values of parameters, the total number of floating point operations (flops) required for each CPI data to be processed throughout this STAP algorithm is 403,552,528. Table 1 shows the number of flops required for each task. A total of 25 CPI complex data cubes were generated as inputs to the parallel pipeline system. Each task in the pipeline contains three major parts: receiving data from the previous task, main computation, and sending results to the next task. Performance results are measured separately for these three parts, namely receiving time, computation time, and sending time. In each task timing results for processing one CPI data were obtained by accumulating the execution time for the middle 20 CPIs and then averaging it. Timing results presented in this paper do not include the effect of initial setup (first 3 CPIs) and final iterations (last 2 CPIs).

7.1 Computation Costs

The task of computing hard weights is the most computationally demanding task. The Doppler filter processing task is the second most demanding task. Naturally, more compute nodes are assigned to these two tasks in order to obtain a good performance. For each task in the STAP algo-

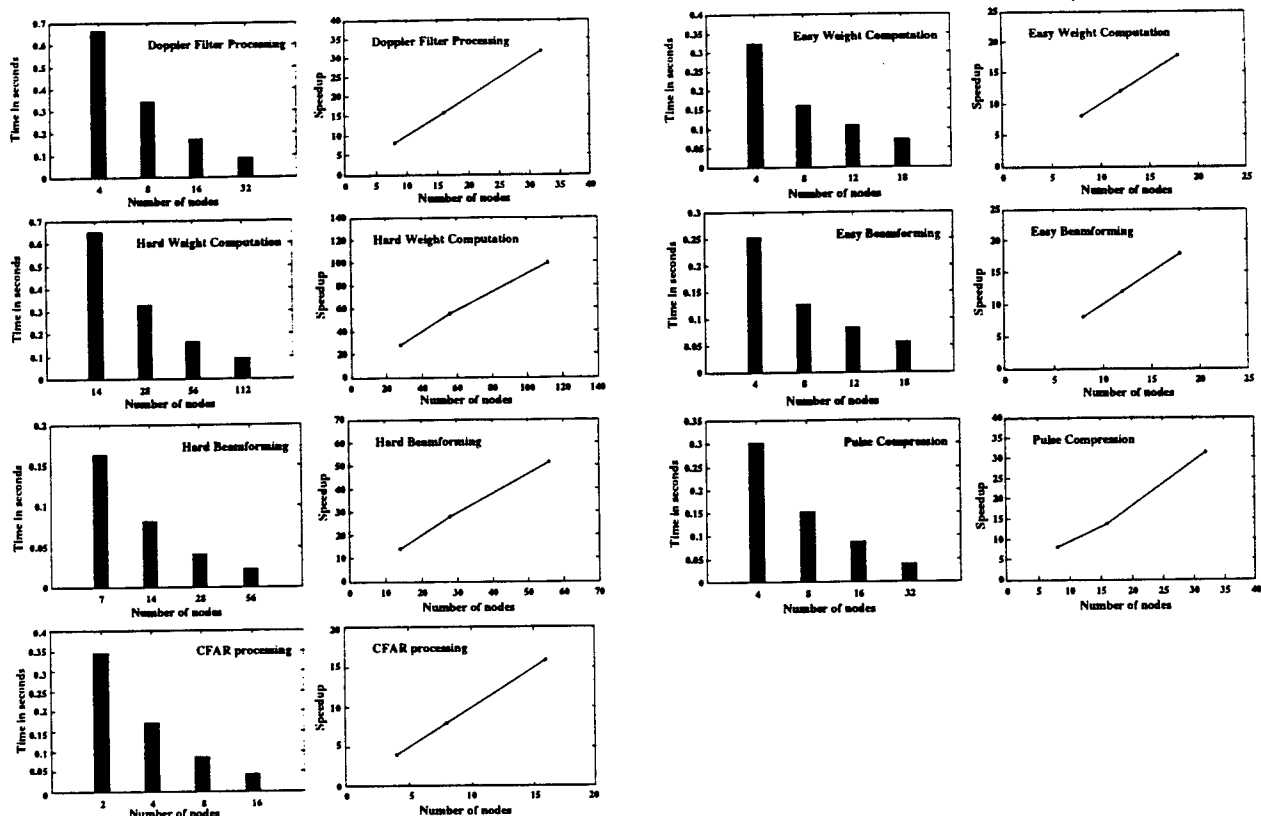


Figure 11. Performance and speedup of computation time as a function of number of compute nodes for all tasks.

algorithm, parallelization was done by evenly dividing computational load across the compute nodes assigned. Since there is no intra-task data dependency, no inter-processor communication occurs within any single task in the pipeline. Another way to view this is that intra-task communication is moved to the beginning of each task within the data redistribution step. Figure 11 gives the computation performance results as functions of numbers of nodes and the corresponding speedup on the AFRL Intel Paragon. For each task, we obtained linear speedups.

7.2 Inter-task Communication

Inter-task communication refers to the communication between sending and receiving (distinct and parallel) tasks. This communication cost depends on both processor assignment for each task as well as on the volume and extent of data reorganization. Tables 2 to 6 present the inter-task com-

Table 2. Timing results of inter-task communication from Doppler filter processing task to its successor tasks. Time in seconds.

	# nodes	easy weight		hard weight				easy BF		hard BF	
		16		56		112		16		16	
Doppler filter		send	recv	send	recv	send	recv	send	recv	send	recv
	8	.1332	.4339	.1332	.3603	.1332	.4441	.1332	.4509	.1332	.4395
	16	.0679	.1780	.0679	.1048	.0679	.1837	.0679	.1955	.0679	.1843
	32	.0340	.0511	.0332	.0034	.0340	.0563	.0340	.0646	.0340	.0519

munication timing results. Each table considers pairs of tasks where the number of compute nodes for both tasks are varied. In some cases timing results shown in the tables contain idle time for waiting for the corresponding task to complete. This happens when receiving task's computation part completes before the sending task has generated data to send.

From most of the results (Tables 2 to 6) the following important observations can be made. First, when the number of nodes is unbalanced (e.g., sending task having small number of nodes while the receiving task has large number of nodes), the communication performance is not very good. Second, as the number of nodes is increased in the sending and receiving tasks, communication scales tremendously. This happens for two reasons. One, each node has less data to reorganize, pack and send and each node has less data to receive; and two, contention at sending and receiving nodes is reduced. For example, Table 2 shows that when the sending task's number of nodes is increased from 8 to 32, the communication times improve in a superlinear fashion. Thus, it is not sufficient to improve the computation times for such parallel pipelined applications to improve throughput and latency.

In Figure 10 receiving time for each loop is given by subtracting t_1 from t_0 . Since computation has to be performed only after input data has been received, receiving time may contain the waiting time for the input, shown in line 4. Sending time, $t_3 - t_2$, measures the time containing data packing (collection and reorganization) and posting sending requests. Because of the asynchronous send used in the implementation, the results shown here are visible sending time and the actual sending action may occur in other portions of the task. Similar to the receiving time, sending

Table 3. Timing results of inter-task communication from easy weight computation task to easy beamforming task. Time in seconds.

	# nodes	easy beamforming			
		8		16	
easy weight		send	recv	send	recv
	4	.0005	.1956	.0007	.2570
	8	.0088	.0883	.0004	.0905
	16	.0768	.0807	.0003	.0660

Table 4. Timing results of inter-task communication from hard weight computation task to hard beamforming task. Time in seconds.

	# nodes	hard beamforming			
		8		16	
hard weight		send	recv	send	recv
	28	.0007	.1798	.0007	.2485
	56	.0100	.1468	.0065	.0765
	112	.1824	.1398	.0005	.0543

time may also contain waiting time for the completion of sending requests in the previous loop, shown in line 8. Especially in the cases when two communicating tasks have uneven partitioned parallel computation load, this effect becomes more apparent. With large number of nodes, there is tremendous scaling in performance of communicating data as the number of nodes is increased. This is because the amount of processing for communication per node is decreased (as it handles less amount of data), amount of data per node to be communicated is decreased and traffic on links going in and out of each node is reduced. This model scales well for both computation and communication.

Table 5. Timing results of inter-task communication from easy and hard beamforming tasks to pulse compression task. Time in seconds.

	# nodes	pulse compression			
		8		16	
		send	recv	send	recv
easy BF	4	.0069	.5016	.0069	.5714
	8	.0036	.1379	.0036	.2090
	16	.0580	.0771	.0022	.0569
		send	recv	send	recv
hard BF	4	.0054	.5016	.0054	.5714
	8	.0029	.1379	.0030	.2090
	16	.1159	.0771	.0017	.0569

Table 6. Timing results of inter-task communication from pulse compression task to CFAR processing task. Time in seconds.

	# nodes	CFAR processing			
		4		8	
		send	recv	send	recv
pulse compr	4	.0099	.3351	.0098	.3348
	8	.0053	.0662	.0051	.1750
	16	.1256	.0435	.0028	.1783

7.3 Integrated System Performance

Integrated system refers to the evaluation of performance when all the tasks are considered together. Throughput and latency are the two most important measures for performance evaluation in addition to individual task computation time and inter-task communication time. Table 7 gives timing results for three different cases with different node assignments.

In section 5 equations (1) and (2) provide the throughput and latency for one CPI data set. The measured throughput is obtained by placing a timer at the end of last task and recording the time

Table 7. Performance results for 3 cases with different node assignments. Time in seconds.

case 1: total number of nodes = 236

	# nodes	recv	comp	send	total
Doppler filter	32	.0055	.0874	.0348	.1276
easy weight	16	.0493	.0913	.0003	.1408
hard weight	112	.0555	.0831	.0005	.1390
easy BF	16	.0658	.0708	.0021	.1387
hard BF	28	.0936	.0414	.0010	.1361
pulse compr	16	.0551	.0776	.0028	.1355
CFAR	16	.0910	.0434	-	.1344
throughput	7.2659				
latency	0.3622				

case 2: total number of nodes = 118

	# nodes	recv	comp	send	total
Doppler filter	16	.0110	.1714	.0668	.2492
easy weight	8	.0998	.1636	.0003	.2637
hard weight	56	.0979	.1636	.0005	.2621
easy BF	8	.1302	.1267	.0036	.2605
hard BF	14	.1782	.0822	.0017	.2622
pulse compr	8	.1027	.1543	.0051	.2621
CFAR	8	.1742	.0864	-	.2606
throughput	3.7959				
latency	0.6805				

case 3: total number of nodes = 59

	# nodes	recv	comp	send	total
Doppler filter	8	.0219	.3509	.1296	.5024
easy weight	4	.1796	.3254	.0003	.5053
hard weight	28	.1779	.3265	.0006	.5050
easy BF	4	.2439	.2529	.0068	.5037
hard BF	7	.3370	.1636	.0032	.5039
pulse compr	4	.1806	.3067	.0097	.4970
CFAR	4	.3240	.1723	-	.4963
throughput	1.9898				
latency	1.3530				

Table 8. Throughput and latency for the 3 cases in Table 7. Real results are obtained from the experiments while equation results are obtained from applying individual tasks' timing to equations (1) and (2). The unit of throughput is number of CPIs per second. The unit of latency is second.

# of nodes		236	118	59
throughput	equation	7.1019	3.7919	1.9791
	real	7.2659	3.7959	1.9898
latency	equation	0.5362	1.0346	1.9996
	real	0.3622	0.6805	1.3530

difference between every loop (that is between two successive completions of the pipeline.) The inverse of this measure provides the throughput. On the other hand, it is more difficult to measure latency because it requires synchronizing clocks at the first task and last task's nodes. Thus, to obtain the measured latency, the timing measurement should be made by first reading time at both first task and last task when the first task is ready to read a new input data. This can be done by sending a signal from the first task to the last task when the first task is ready for reading the new input data. Then the timer for last task can be started.

In fact, the latency given in equation (2) represents an **upper bound** because the way we time tasks contains the time of waiting for input from previous task. This waiting time portion overlaps with the computation time in the previous tasks and should be excluded from the latency. Thus the latency results are conservative values and the real latency is expected to be smaller than this value. However, the latency given from equation (2) indicates the worst-case performance for our implementation. The real latency equation, therefore, becomes

$$real\ latency = T_0 + \max(T'_3, T'_4) + T'_5 + T'_6 \quad (3)$$

where $T'_i = T_i$ - idle time at receiving, $i = 3, 4, 5$, and 6 .

Table 8 gives the throughput and latency results for the 3 cases shown in Table 7. From these 3 cases, it is clear that even for latency and throughput measures we obtain linear speedups from our experiments. Given that this scale up is up to compute 236 nodes (we were limited to these

Table 9. Performance results for adding 4 more nodes to Doppler filter processing task to case 2 in Table 7. Time in seconds.

total number of nodes = 122					
	# nodes	recv	comp	send	total
Doppler filter	20	.0090	.1395	.0540	.2024
easy weight	8	.0519	.1633	.0003	.2155
hard weight	56	.0486	.1644	.0005	.2135
easy BF	8	.0815	.1272	.0037	.2124
hard BF	14	.1232	.0823	.0018	.2073
pulse compr	8	.0519	.1543	.0051	.2113
CFAR	8	.1240	.0864	-	.2105
throughput	5.0213				
latency	0.5498				

number of nodes due to the size of the machine), we believe these are very good results.

As discussed in section 4, tradeoffs exist between assigning nodes to maximize throughput and to minimize latency, given limited resources. Using two examples, we illustrate how further performance improvements may (or may not) be achieved if few extra nodes are available. We now take case 2 from Table 7 as an example and add some extra nodes to tasks to analyze its affect to the throughput and latency. Suppose that case 2 has fulfilled the minimum throughput requirement and more nodes can be added. Table 9 shows that adding 4 more nodes to Doppler filter processing task not only increases the throughput but also reduces the latency. This is because the communication amount for each send and receive between Doppler filter processing task to weight computation and to beamforming tasks is reduced (Table 9). So, clearly adding nodes to one task not only affects that task's performance but has a measurable effect on the performance of other tasks. By increasing the number of nodes 3%, the improvement in throughput is 32% and in latency is 19%. *Such effects are very difficult to capture in purely theoretical models because of the secondary effects.*

Since the parallel computation load may be different among tasks, bottleneck problems arise when some tasks in the pipeline do not have proper numbers of nodes assigned. If the number of

Table 10. Performance results for adding 16 more nodes to pulse compression and CFAR processing tasks to the case in Table 9. Time in seconds.

total number of nodes = 138

	# nodes	recv	comp	send	total
Doppler filter	20	.0091	.1395	.0541	.2027
easy weight	8	.0516	.1633	.0003	.2152
hard weight	56	.0488	.1644	.0005	.2137
easy BF	8	.0819	.1273	.0037	.2129
hard BF	14	.1301	.0823	.0018	.2142
pulse compr	16	.1337	.0775	.0028	.2140
CFAR	16	.1701	.0434	-	.2135
throughput	4.9052				
latency	0.4247				

nodes assigned to one task with heavy work load is not enough to catch up the input data rate, this task becomes a bottleneck in the pipeline system. Hence, it is important to maintain approximately the same computation time among tasks in the pipeline system to maximize the throughput and also achieve higher processor utilization. One bottleneck task can be seen when its computation time is relatively much larger than the rest of the tasks. The entire system's performance degrades because the rest of the tasks have to wait for bottleneck task's completion to send/receive data to/from it no matter how many more nodes assigned to them and how fast they can complete their jobs. Therefore, poor task scheduling and processor assignment will cause significant portion of idle time in the resulted communication costs. In Table 10 we added a total of 16 more nodes to pulse compression and CFAR processing tasks to the case in Table 9. Comparing to case 2 in Table 7, we can see that the throughput increased. However, the throughput did not improve compared to the results in Table 9, even though this assignment has 16 more nodes. In this case, the weight tasks are bottleneck tasks because their computation costs are relatively higher than other tasks. We can see that the receiving time of the rest of tasks are much larger than their computation time. A significant portion of idle time waiting for the completion of weight tasks is in the receiving time. On the other hand, we observe 23% improvement in the latency. This is because the computation

time is reduced in the last two tasks with more nodes assigned. From equation (3), the execution time of these two tasks, T'_5 and T'_6 , decreases and therefore the latency is reduced.

8 Conclusions

In this paper we presented performance results for a PRI-staggered post-Doppler STAP algorithm implementation on the Intel Paragon machine at Air Force Research Laboratory, Rome, New York. The results indicate that our approach of parallel pipelined implementation scales well both in terms of communication and computation. For the integrated pipeline system, the throughput and latency also demonstrate the linear scalability of our design. Linear speedups were obtained for up to 236 compute nodes. When more than 236 nodes are used, the speedup curves for the results of throughput and latency may saturate. This is because the communication costs will become significant with respect to the computation costs.

Almost all radar applications have real-time constraints, hence a well designed system should be able to handle any changes in the requirements on the response time by dynamically allocating or re-allocating processors among tasks. Our design and implementation not only shows tradeoffs in parallelization, processor assignment, and various overheads in inter and intra-task communication etc., but it also shows that accurate performance measurement of these systems is very important. Consideration of issues such as cache performance when data is packed and unpacked, and impact of the parallelization and processor assignment for one task on another task are crucial. This is normally not easily captured in theoretical models. In the future we plan to incorporate further optimizations including multi-threading, multiple pipelines and multiple processors on each compute node.

9 Acknowledgments

This work was supported by Air Force Materiel Command under contract F30602-97-C-0026. We acknowledge the use of the Intel Paragon at Caltech for initial development.

References

- [1] M. Linderman and R. Linderman, "Real-Time STAP Demonstration on an Embedded High Performance Computer," *IEEE AES Systems Magazine*, pp. 15–21, Mar. 1998.
- [2] R. Brown and R. Linderman, "Algorithm Development for an Airborne Real-Time STAP Demonstration," in *Proceedings of the IEEE National Radar Conference*, 1997.
- [3] M. Little and W. Berry, "Real-Time Multi-Channel Airborne Radar Measurements," in *Proceedings of the IEEE National Radar Conference*, 1997.
- [4] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin-Cummings, 1994.
- [5] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors*, Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [6] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, Inc., 1993.
- [7] G. Golub and J. Ortega, *Scientific Computing: An Introduction with Parallel Computing*, Academic Press, Boston, MA, 1993.
- [8] C. Xavier and S. Iyengar, *Introduction to Parallel Algorithms*, John Wiley & Sons, Inc., 1998.
- [9] J. Lebak, R. Durie, and A. Bojanczyk, "Toward A Portable Parallel Library for Space-Time Adaptive Methods," Tech. Rep. CTC96TR242, Cornell Theory Center, June 1996.
- [10] S. Olszanskyj, J. Lebak, and A. Bojanczyk, "Parallel Algorithms for Space-Time Adaptive Processing," *International Parallel Processing Symposium*, pp. 77–81, Apr. 1995.
- [11] Y. Lim and V. Prasanna, "Scalable Portable Implementations of Space-Time Adaptive Processing," in *Proceedings of the 10th International Conference on High Performance Computing*, June 1996.

- [12] P. Bhat, Y.Lim, and V. Prasanna, "Issues in using Heterogeneous HPC Systems for Embedded Real Time Signal Processing Applications," in *Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, Oct. 1995.
- [13] M. Lee and V. Prasanna, "High Throughput-Rate Parallel Algorithms for Space Time Adaptive Processing," *2nd International Workshop on Embedded Systems and Applications*, Apr. 1997.
- [14] D. Martinez, "Application of Parallel Processors to Real-Time Sensor Array Processing," *International Parallel Processing Symposium*, Apr. 1999.
- [15] K. Cain, J. Torres, and R. Williams, "RT-STAP: Real-Time Space-Time Adaptive Processing Benchmark," Tech. Rep. 96B0000021, MITRE Corporation, Feb. 1997.
- [16] C. Brown, M. Flanzbaum, R. Games, and J. Ramsdell, "Real-Time Embedded High Performance Computing: Application Benchmarks," Tech. Rep. MTR94B145, MITRE Corporation, Oct. 1994.
- [17] A. Choudhary and J. Patel, *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems*, Kluwer Academic Publishers, Boston, MA, 1990.
- [18] A. Choudhary and R. Ponnusamy, "Run-Time Data Decomposition for Parallel Implementation of Image Processing and Computer Vision Tasks," *Journal of Concurrency, Practice and Experience*, vol. 4, no. 4, pp. 313–334, June 1992.
- [19] A. Choudhary and R. Ponnusamy, "Parallel Implementation and Evaluation of a Motion Estimation System Algorithm using Several Data Decomposition Strategies," *Journal of Parallel and Distributed Computing*, vol. 14, pp. 50–65, January 1992.
- [20] R. Thakur, A. Choudhary, and J. Ramanujam, "Efficient Algorithms for Array Redistribution," *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 7, pp. 587–594, June 1996.
- [21] M. Berger and S. Bokhari, "A Partitioning Strategy for Nonuniform Problems on Multiprocessors," *IEEE Trans. on Computers*, vol. 36, no. 5, pp. 570–580, May 1987.

I/O Implementation and Evaluation of Parallel Pipelined STAP on High Performance Computers

Wei-keng Liao[†], Alok Choudhary[‡], Donald Weiner[†], and Pramod Varshney[†]

[†] EECS Department
Syracuse University
Syracuse, NY 13244

[‡] ECE Department
Northwestern University
Evanston, IL 60208

Abstract

This paper presents experimental results for a parallel pipeline STAP system with I/O task implementation. In our previous work, a parallel pipeline model was designed for radar signal processing applications on parallel computers. Based on this model, we implemented a real STAP application which demonstrated the performance scalability of this model in terms of throughput and latency. The parallel pipeline model normally does not include I/O task because the input data can be provided directly from radars. However, I/O can also be done through disk file systems if radar data is stored in disks first. In this paper, we study the effect on system performance when the I/O task is incorporated in the parallel pipeline model. There are two alternatives for I/O implementation: embedding I/O in the pipeline or having a separate I/O task. We used the parallel file systems on the Intel Paragon and the IBM SP to perform parallel I/O and studied its effects on the overall performance of the pipeline system. From these two I/O implementations, we discovered that the latency may be improved when the structure of the pipeline is reorganized by merging multiple tasks into a single task. Finally, we investigated the problem of data redistribution embedded in the parallel I/O when

special hardware is not available to pre-process the raw signal data before it enters the pipeline system. All the performance results shown in this paper demonstrated the scalability of parallel I/O implementation on the parallel pipeline STAP system.

1 Introduction

In this paper we build upon our earlier work where we devised strategies for high performance parallel pipeline implementations, in particular, for Space-Time Adaptive Processing (STAP) applications [2, 8]. A modified Pulse Repetition Interval (PRI)-staggered post-Doppler STAP algorithm was implemented based on the parallel pipeline model and scalable performance was obtained both on the Intel Paragon and the IBM SP. Normally, this parallel pipeline system does not include disk I/O costs. Since most radar applications require signal processing in real time, thus far we have assumed that the signal data collected by radar is directly delivered to the pipeline system, as shown in the overall radar and signal processing system of Figure 1.

In practice, the I/O can be done either directly from a radar or through disk file systems.

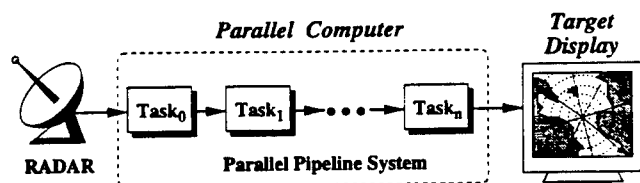


Figure 1. Data flow of a radar and signal processing system using parallel computers.

In this work we focus on the I/O implementation of the parallel pipeline STAP algorithm when I/O is carried out through a disk file system. Using existing parallel file systems, we investigate the impact of I/O on the overall pipeline system performance. Two designs of I/O are employed: in the first design the I/O is embedded in the pipeline without changing the task structure and in the other a separate task is created to perform I/O operations. With different I/O strategies, we ran the parallel pipeline STAP system portably and measured the performance on the Intel Paragon at California Institute of Technology and on the IBM SP at Argonne National Laboratory (ANL.) The parallel file systems on both the Intel Paragon and the IBM SP contain multiple stripe directories for applications to access disk files efficiently. On the Paragon, two PFS file systems with different stripe factors were tested and the results were analyzed to assess the effects of the size of the stripe factor on the STAP pipeline system. On the IBM SP, the performance results were obtained by using the native parallel file system, PIOFS, which has 80 stripe directories.

Comparing the two parallel file systems with different stripe sizes on the Paragon, we found that an I/O bottleneck results when a file system with smaller stripe size is used. Once a bottleneck appears in a pipeline, the throughput which is determined by the task with maximum execution time degrades significantly. On the other hand, the latency is not significantly affected by the bottleneck problem.

This is because the latency depends on all the tasks in the pipeline rather than the task with the maximum execution time. Furthermore, when evaluating the performance results of the two I/O designs, we observed that the latency can be improved by merging two tasks in the pipeline. In this paper, we also examine the possibility of improving latency by reorganizing the task structure of the STAP pipeline system.

A sequence of raw signal data sets collected by a radar form the input to the STAP pipeline system. Each of these raw data sets is in the form of a three dimensional array. However, the three dimensions of this array are not organized in a way such that each Fast Fourier Transformation (FFT) in the Doppler filter processing task can be performed in a single processor. Without special hardware support to pre-process the collected raw data, data redistribution is needed before delivering the data to the Doppler filter processing task. In the real application we implemented, this pre-processing work includes data type conversion and corner turn on the three-dimensional array. Using a software approach, we also embedded pre-processing operation on the raw data in the two I/O designs and compared their performances.

The rest of the paper is organized as follows: in Sections 2 and 3, we briefly describe our previous work, the parallel pipeline system model and its implementation on a STAP algorithm. The characteristics of the parallel file systems tested in this paper are described in Section 4. The I/O design and implementation are presented in Section 5 and their performance results are given in Section 6. Section 7 presents the implementation when tasks are combined to improve latency. The software approach to pre-processes raw signal data is described in Section 8. Conclusions are given in Section 9.

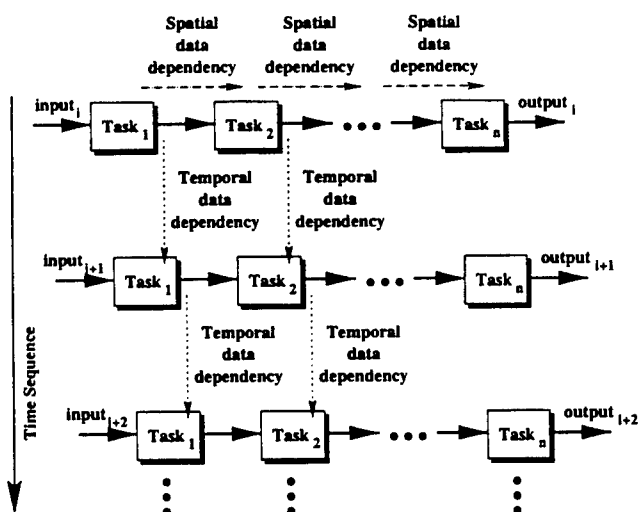


Figure 2. Model of the parallel pipeline system. The set of pipelines indicates that the same pipeline is repeated on subsequent input data sets. Each task for all input instances is executed on the same number of compute nodes.

2 Model of the parallel pipeline system

The system model for the type of STAP applications considered in this work is shown in Figure 2. A pipeline is a collection of tasks which are executed sequentially. The input to the first task is obtained normally from sensors or other input devices and the inputs to the rest of the tasks in the pipeline are the outputs of their previous tasks. The set of pipelines shown in the figure indicates that the same pipeline is repeated on subsequent input data sets. Each block in a pipeline represents one parallel task, which itself is parallelized on multiple compute nodes.

2.1 Data dependency

In such a parallel pipeline system, there exist both spatial and temporal parallelism that result in two types of data dependencies, namely,

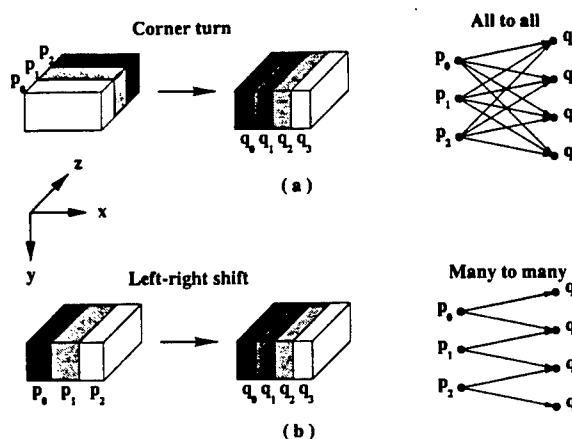


Figure 3. Two types of data redistribution: corner turn and left-right shift. Corner turn involves an all-to-all personalized communication and left-right shift involves a many-to-many communication.

spatial data dependency and temporal data dependency [3, 4]. Spatial data dependency can be classified into inter-task data dependency and intra-task data dependency. Intra-task data dependencies arise when a set of subtasks needs to exchange intermediate results during the execution of a parallel task in a pipeline. Inter-task data dependency is due to the transfer and reorganization of data passed onto the next parallel task in the pipeline. Temporal data dependency occurs when some form of output generated by the tasks executed on the previous data set are needed by tasks executing the current data set. We will later see that STAP has both types of data dependencies.

2.2 Data redistribution

In the parallel pipeline system shown in Figure 2, compute nodes are partitioned into several disjoint groups and each group is assigned to exactly one task in the pipeline. Data transfer between two tasks represents interprocessor communication between two groups of compute nodes. Since the data access patterns of one task may be different from its successor

tasks, communication patterns can be either a corner-turn or a left-right shift pattern, shown in Figure 3.

Given a three-dimensional array as a pipeline system input, the data partitioning can be done along one of the array's three axes. Any single data layout will not always provide efficient computation for data access along two orthogonal axes. This communication pattern is called a corner-turn communication. The corner-turn communication involves a complete exchange (all-to-all personalized) of data between two groups of compute nodes. The left-right shift communication pattern occurs when an array is partitioned along the same axis between two consecutive parallel tasks. It does not involve data reorganization and each node in one task only communicates with some of the nodes in its successor task (a many-to-many communication.)

3 Parallel pipeline STAP system

In our previous work [2], we described the parallel pipelined implementation of a PRI-staggered post-Doppler STAP algorithm. This STAP algorithm consists of five steps: 1) Doppler filter processing, 2) weight computation, 3) beamforming, 4) pulse compression, and 5) CFAR processing. The design of the parallel pipelined STAP algorithm is shown in Figure 4. The parallel pipeline system consists of seven tasks. Both the weight computation and the beamforming tasks are divided into two parts, namely, "easy" and "hard" Doppler bins. The hard Doppler bins are those in which significant ground clutter is expected and the remaining bins are easy Doppler bins. The main difference between the two is the amount of data used and the amount of computation required.

The input data set for the pipeline is obtained from a phased array radar and is formed in terms of a coherent processing interval

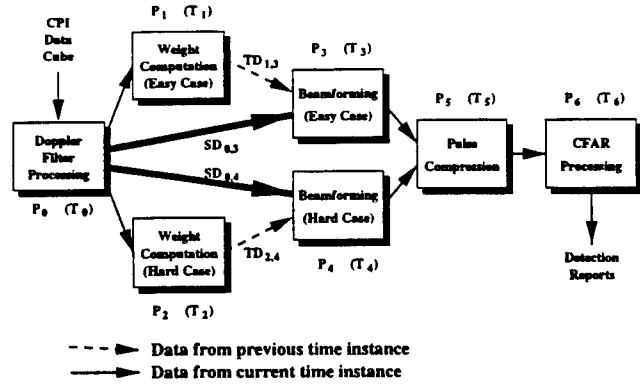


Figure 4. Implementation of parallel pipelined STAP. Arrows connecting task blocks represent data transfer between tasks.

(CPI). Each CPI data set is a 3-dimensional complex data cube. The output of the pipeline is a report on the detection of possible targets. Each task i , $0 \leq i < 7$, is parallelized by evenly partitioning its work load among P_i compute nodes. The execution time associated with task i is T_i . For the computation of the weight vectors for the current CPI data cube, data cubes from previous CPIs are used as input data. This introduces temporal data dependency. Temporal data dependencies are represented by arrows with dashed lines in Figure 4 where $TD_{i,j}$ represents temporal data dependency of task j on data from task i . In a similar manner, spatial data dependencies $SD_{i,j}$ can be defined and are indicated by arrows with solid lines.

Throughput and latency are two important measures for performance evaluation on a pipeline system.

$$throughput = \frac{1}{\max_{0 \leq i < 7} T_i}. \quad (1)$$

$$latency = T_0 + \max(T_3, T_4) + T_5 + T_6. \quad (2)$$

The temporal data dependency does not affect the latency because weight computation tasks use data from the previous time instance rather than the current CPI. The filtered CPI

data cube sent to the beamforming task does not wait for the completion of its weight computation. This explains why Equation (2) does not contain T_1 and T_2 . A detailed description of the STAP algorithm we used can be found in [1, 9].

4 Parallel file systems

Only input part of parallel I/O was implemented on the STAP pipeline system because most applications like STAP send their detection results to display devices in real time. The input to the STAP pipeline system is a series of CPI data sets captured by the radar. To test our parallel pipeline system with regard to I/O performance, these CPI data sets were stored in the parallel file system and provided to the pipeline system through machine's I/O nodes. We used the parallel I/O library developed by Intel Paragon and IBM SP systems to perform read operations.

4.1 Intel Paragon PFS file system

The Intel Paragon OSF/1 operating system provides a special file system type called PFS, for Parallel File System, which gives applications high-speed access to a large amount of disk storage [7]. PFS file systems are optimized for simultaneous access by multiple nodes. Each PFS file system consists of multiple stripe directories. Each stripe directory is the mount point of a separate UNIX file system. A PFS file system collects together several file systems into a unit that behaves like a single large file system. A file stored in PFS is distributed, or striped, across the stripe directories that make up the PFS file system. The performance of accessing a single PFS file is significantly improved by multiple stripe devices providing disk data simultaneously. The amount of data from a PFS file that is stored in each stripe directory is determined by the

PFS file system's stripe unit. The stripe units on all Paragon parallel systems at Caltech are 64K bytes. Two PFS file systems were tested: one has 16 stripe directories (stripe factor 16) and the other has a stripe factor of 64.

We used the Intel Paragon NX library to implement the I/O of the parallel pipeline STAP system. Since only input part of the I/O is needed for providing a series of CPI data sets to the pipeline, only read operations are investigated. Subroutine **gopen()** was used to open CPI files globally because it offers better performance and causes less system overhead. NX library provides six I/O modes for an application to access files: **M_UNIX**, **M_LOG**, **M_SYNC**, **M_RECORD**, **M_GLOBAL**, and **M_ASYNC**. A file's I/O mode is set when the file is opened with **gopen()**. Only non-collected I/O mode **M_ASYNC** was used because it provided an efficient parallel read operation. This mode has the following characteristics on an opened PFS file:

- every node has its own file pointer
- read operations are not synchronized
- read can be for variable-length, unordered records

This mode allows multiple reads to access a single file simultaneously without agreement on record size or file offset among nodes. If read operations access exclusive portions of a file, it behaves like each compute node reads from its own file independently. In the pipeline system, the number of nodes to read CPI files may vary and, therefore, the length of the subset of CPI file for each node to read can be different. Besides, only the nodes in the first task of the pipeline system issue read operations, rather than all nodes allocated for the whole application. This explains why we used **M_ASYNC** mode and it is also the only feasible and efficient way to read disk files in parallel. All other collective I/O modes provided by the

OSF/1 operating system require that all nodes in the application perform the same I/O operations and, hence, accessing files by a subset of the nodes is prohibited for these modes. In addition, we used asynchronous I/O function calls: `iread()` and `ireadoff()` in order to overlap I/O operations with the computation and communication.

4.2 IBM SP PIOFS file system

The IBM AIX operating system provides a parallel file system called Parallel I/O File System (PIOFS) which is designed for IBM RS/6000 SP to allow fast parallel access to large temporary data files [6]. The PIOFS on the IBM SP at ANL is made up of 5 servers. Four of the servers have 4 Serial Storage Architecture (SSA) disks while the fifth is the directory server. Each of the 4 SSA disks is partitioned into 5 slices. Therefore, there are a total of 80 slices (striped directories) in the ANL PIOFS file system. The default basic striped unit (BSU) is 64K bytes. A file stored in the PIOFS is physically divided into several blocks with each equal to the size of one BSU, and these blocks are stored in the 80 striped directories in a round-robin manner.

IBM PIOFS supports existing C read, write, open and close functions. In addition to a UNIX-like I/O interface, PIOFS also supports logical partitioning of files. A processor can independently specify a logical view of the data in a file, a subfile, and then perform I/O on this subfile with a single call. In our STAP I/O task implementation, we store all CPI files in the ANL PIOFS using the default BSU, 64K bytes. As for the Intel Paragon, CPI files are stored across 80 striped directories in the PIOFS file system. However, unlike the Paragon NX library, asynchronous parallel read/write subroutines are not supported on IBM PIOFS. The overall performance of the STAP pipeline system will be limited by the inability to over-

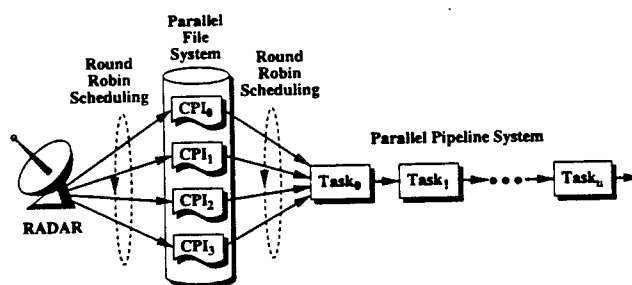


Figure 5. Four CPI data files are read from the parallel file system into the pipeline system in a round-robin manner.

lap I/O operations with computation and communication.

5 Design and implementation

A total of four CPI data sets stored as four files in the parallel file systems were used on both the Caltech Paragon and the ANL SP. Each of the four CPI files is of size 8M bytes. On the Paragon, these files are opened globally (or collectively) by all compute nodes allocated in the whole application during the STAP pipeline system's initialization. On SP, these four files are opened only by the compute nodes that perform the I/O task. During each of the following steps after the initialization, only nodes assigned to the first task perform read operations from the parallel file system. We assume that the radar writes its collected CPI data into these four files in a round-robin manner. Similarly, the STAP pipeline system was also designed to read these four files in a round-robin fashion but at times that are different from the times at which the radar writes. This is shown in Figure 5. In this manner, the problem of data inconsistency for read/write operations between the radar and the STAP parallel pipeline system is minimized.

All nodes allocated to the first task (the I/O nodes) of the pipeline read exclusive portions of each CPI file with proper offsets. Because

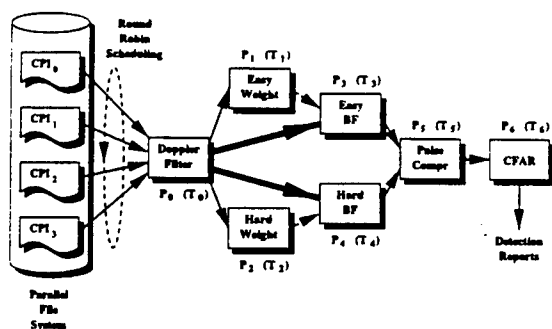


Figure 6. I/O task is embedded in the Doppler filter processing task of the STAP pipeline system.

the number of I/O nodes may vary due to different node assignments to the I/O task, the length of data for the read operations can be different. The read length and file offset for all the read operations are set only during the STAP pipeline system's initialization and is not changed afterward. Therefore, in each of the following iterations, only one read function call is needed. On the Paragon, since asynchronous read subroutines were used, an additional subroutine waiting for the read's completion was also required in each iteration.

5.1 I/O task implementation

Two designs for the I/O task were implemented in the STAP pipeline system. The first one, shown in Figure 6, embeds the parallel I/O in the first task of the pipeline, i.e. in the Doppler filter processing task. The Doppler filter processing task now consists of three phases, reading CPI data from files, computation, and sending phases. The second I/O implementation creates a new task for reading CPI data and this task is added to the beginning of the pipeline. Figure 7 shows the structure of the overall pipeline system with this implementation. The only job of this I/O task is to read CPI data from the files and deliver it to the Doppler filter processing task.

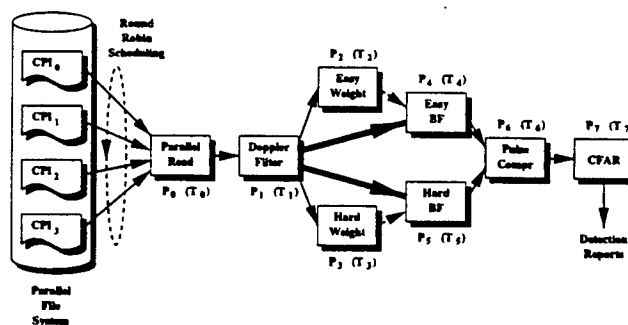


Figure 7. A separate I/O task for reading CPI data is added to the STAP pipeline system.

6 Performance results

Performance results are given for the two I/O implementations on the parallel pipeline STAP system. For each implementation, parallel file systems on the Paragon and the SP were tested. On the Paragon, we used two PFS file systems, one with 16 stripe directories and the other with 64 stripe directories. On the SP, only the parallel file system with 80 striped directories was tested. On both machines, the stripe unit for the parallel file systems is 64K bytes. The size of each CPI data file is 8M bytes that results in 128 stripe units distributed across all stripe directories in all the parallel file systems.

6.1 I/O embedded in the first task

In the first I/O implementation on the Paragon, the Doppler filter processing task reads its input from CPI files using asynchronous read calls. A double buffering strategy is employed to overlap the I/O operations with computation and communication in this task. Table 1 shows the timing results for this implementation on the Paragon PFS file system with 16 stripe directories. Three cases of node assignments to all tasks in the pipeline system are given, each doubles the number of nodes of another. The throughput scales well in the first two cases, but degrades when the

Table 1. Performance results on the Paragon with the I/O embedded in the Doppler filter processing task.

PFS stripe factor = 16

Table 2. Performance results on the Paragon with the I/O embedded in the Doppler filter processing task.

PFS stripe factor = 64

case 1: total number of nodes = 56 Time in seconds

	# nodes	recv	comp	send	total
Doppler filter	12	.0101	.2566	.0916	.3584
easy weight	3	.1317	.2214	.0002	.3534
hard weight	28	.0684	.2838	.0003	.3525
easy BF	3	.1451	.1921	.0003	.3375
hard BF	4	.1596	.1756	.0002	.3354
pulse compr	4	.1070	.1979	.0298	.3347
CFAR	2	.1983	.1361	-	.3343
throughput			2.9560		
latency			0.9804		

case 2: total number of nodes = 112 Time in seconds

	# nodes	recv	comp	send	total
Doppler filter	24	.0178	.1292	.0663	.2134
easy weight	6	.0856	.1110	.0002	.1968
hard weight	56	.0483	.1423	.0059	.1965
easy BF	6	.0939	.0958	.0003	.1901
hard BF	8	.0906	.0885	.0003	.1795
pulse compr	8	.0648	.0993	.0150	.1792
CFAR	4	.1107	.0683	-	.1790
throughput			5.4996		
latency			0.5171		

case 3: total number of nodes = 224 Time in seconds

	# nodes	recv	comp	send	total
Doppler filter	48	.0871	.0619	.0317	.1807
easy weight	12	.1056	.0557	.0002	.1616
hard weight	112	.0905	.0724	.0009	.1639
easy BF	12	.1080	.0482	.0003	.1565
hard BF	16	.1030	.0509	.0003	.1542
pulse compr	16	.0983	.0502	.0078	.1562
CFAR	8	.1217	.0343	-	.1561
throughput			6.2708		
latency			0.3292		

case 1: total number of nodes = 56 Time in seconds

	# nodes	recv	comp	send	total
Doppler filter	12	.0314	.2461	.0916	.3691
easy weight	3	.1262	.2216	.0002	.3480
hard weight	28	.0628	.2840	.0003	.3471
easy BF	3	.1397	.1921	.0003	.3321
hard BF	4	.1537	.1756	.0002	.3295
pulse compr	4	.1011	.1977	.0298	.3286
CFAR	2	.1920	.1363	-	.3282
throughput			3.0111		
latency			0.9787		

case 2: total number of nodes = 112 Time in seconds

	# nodes	recv	comp	send	total
Doppler filter	24	.0107	.1280	.0557	.1944
easy weight	6	.0787	.1111	.0020	.1917
hard weight	56	.0453	.1427	.0039	.1919
easy BF	6	.0860	.0959	.0003	.1823
hard BF	8	.0878	.0885	.0003	.1766
pulse compr	8	.0615	.0995	.0151	.1761
CFAR	4	.1077	.0682	-	.1759
throughput			5.6068		
latency			0.5143		

case 3: total number of nodes = 224 Time in seconds

	# nodes	recv	comp	send	total
Doppler filter	48	.0069	.0673	.0309	.1052
easy weight	12	.0510	.0559	.0002	.1071
hard weight	112	.0355	.0733	.0019	.1106
easy BF	12	.0526	.0483	.0003	.1013
hard BF	16	.0471	.0515	.0003	.0989
pulse compr	16	.0407	.0503	.0080	.0990
CFAR	8	.0642	.0343	-	.0985
throughput			10.0262		
latency			0.2871		

Table 3. Performance results on the SP with the I/O embedded in the Doppler filter processing task.

PIOFS stripe factor = 80

case 1: total number of nodes = 18					
	# nodes	recv	comp	send	total
Doppler filter	6	.1172	.0734	.1966	.3872
easy weight	1	.2717	.1070	.0001	.3788
hard weight	7	.1590	.2194	.0002	.3786
easy BF	1	.2927	.0829	.0001	.3757
hard BF	1	.2595	.1177	.0002	.3775
pulse compr	1	.2230	.1545	.0001	.3776
CFAR	1	.2941	.0828	-	.3770
throughput				2.6715	
latency				1.2353	

case 2: total number of nodes = 30					
	# nodes	recv	comp	send	total
Doppler filter	8	.1109	.0543	.1031	.2683
easy weight	1	.1471	.1045	.0002	.2518
hard weight	14	.1523	.1072	.0002	.2597
easy BF	2	.2189	.0412	.0001	.2602
hard BF	2	.1999	.0606	.0001	.2606
pulse compr	2	.1801	.0777	.0001	.2579
CFAR	1	.1801	.0801	-	.2602
throughput				3.8319	
latency				0.7810	

case 3: total number of nodes = 60					
	# nodes	recv	comp	send	total
Doppler filter	16	.1044	.0304	.0474	.1823
easy weight	2	.1314	.0547	.0001	.1862
hard weight	28	.1303	.0566	.0002	.1871
easy BF	4	.1571	.0219	.0002	.1792
hard BF	4	.1492	.0298	.0002	.1792
pulse compr	4	.1370	.0396	.0001	.1767
CFAR	2	.1399	.0403	-	.1802
throughput				5.5364	
latency				0.5004	

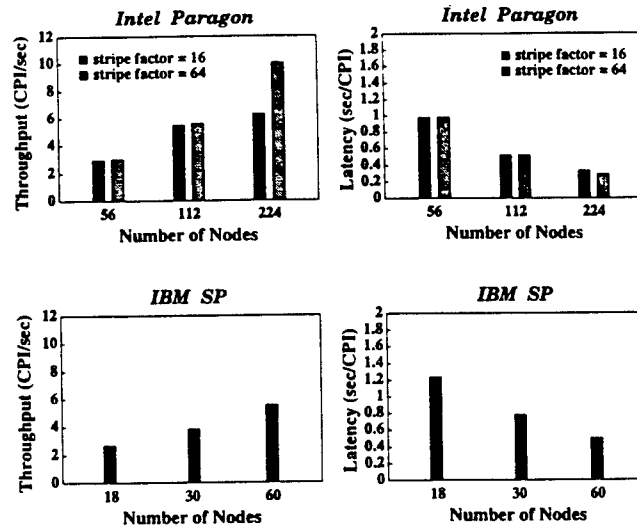


Figure 8. Performance results for the STAP pipeline system with parallel I/O embedded in the Doppler filter processing task. This figure corresponds to Tables 1, 2, and 3.

total number of nodes goes up to 224. In this case, we observe that the timing results of the receive phase in the first task are relatively higher than the other two phases, the compute and send phases. The I/O operations for reading CPI data files here become a bottleneck for the pipeline system. This bottleneck forces the rest of the following tasks in the pipeline system to wait for their input data from their previous tasks.

Table 2 gives the timing results for the same cases as in Table 1, but on a Paragon PFS file system with 64 stripe directories. Both throughput and latency showed linear speedups. In the first two cases with 56 and 112 nodes, the results of throughput and latency are approximately the same for both file systems with 16 and 64 stripe directories. However, in the case with 224 nodes, we observe that the I/O bottleneck is relieved by using 64 stripe directories. The efficiency of I/O operations plays an important role in the overall performance of the pipeline system. The I/O task may become a bottleneck in the

pipeline and directly affect the throughput results.

On the other hand, a linear speedup was obtained for the latency results. The I/O bottleneck problem does not affect the latency significantly. We can observe that in the case with 224 nodes, the latency of using 16 stripe directories is slightly greater than using 64 stripe directories. This can be explained by examining the throughput and latency equations, (1) and (2), shown in Section 3. Unlike the throughput that depends on the maximum of the execution times of all the tasks, the latency is determined by the sum of the execution times of all the tasks except for the tasks with temporal dependency. Therefore, even though the execution time of the Doppler filter processing task is increased, the delay does not contribute much to the latency. Comparing Tables 1 and 2, the latency did not degrade significantly and still scaled well in the case with 224 nodes. Figure 8 shows the performance results of this I/O design in bar charts.

Detailed timing results for the IBM SP at ANL are given in Table 3. The stripe factor of the PIOFS file system is 80. Because PIOFS does not provide asynchronous read/write subroutines, the I/O operations do not overlap with computation and communication in the Doppler filter processing task. Hence, the performance results for throughput and latency on the SP did not show the scalability as on the Paragon, even though the SP has faster CPUs.

6.2 A new I/O task

In the second I/O task implementation, a new task is added to the beginning of the pipeline. This new task only performs the operations of reading CPI files and distributing CPI data to its successor task, Doppler filter processing task. The STAP pipeline system then has a total of 8 tasks. Tables 4, 5, and 6 show the performance results for this

I/O design. Corresponding to Tables 1, 2, and 3, all tasks have the same numbers of nodes assigned, except for the I/O task. The I/O bottleneck problem still occurs when using the Paragon PFS system with 16 stripe directories. When using the file system with 64 stripe directories, the throughput results improved. The bar charts shown in Figure 9 represent the throughput and latency results of Tables 4, 5, and 6.

Comparing the two I/O designs, we observe that the throughput results are approximately the same for both implementations. However, the latency results for the separate I/O task design are worse than the embedded implementation. This phenomenon can be explained by examining the throughput and latency equations. The equations for the throughput and latency for the STAP pipeline system are

$$throughput_8 = \frac{1}{\max_{0 \leq i < 8} T_i} \quad (3)$$

and

$$latency_8 = T_0 + T_1 + \max(T_4, T_5) + T_6 + T_7, \quad (4)$$

where T_i is the execution time for the task i .

The throughput of a pipeline system is determined by the task with the maximum execution time among all the tasks. From Tables 4 and 5, we observe that the Doppler filter processing task has the maximum execution time among all the tasks in the cases with a total of 60 and 120 nodes. In the case of 240 nodes on the PFS file system with 16 stripe directories, the maximum execution time occurs in the parallel I/O task. Using PFS with 64 stripe directories, the hard weight computation task has the maximum execution time in the case of 240 nodes. Compared to Tables 1 and 2, the throughput results have no significant change because the tasks with the maximum execution time are the same for every corresponding pair in all cases. All these tasks have the

Table 4. Performance results on the Paragon with the I/O implemented as a separate task.

PFS stripe factor = 16

case 1: total number of nodes = 60					
	# nodes	recv	comp	send	total
Parallel read	4	.0191	-	.3997	.4187
Doppler filter	12	.0122	.3240	.2375	.5738
easy weight	3	.2032	.2217	.0002	.4252
hard weight	28	.1390	.2846	.0003	.4239
easy BF	3	.2210	.1911	.0003	.4124
hard BF	4	.2327	.1753	.0003	.4083
pulse compr	4	.1800	.1977	.0295	.4072
CFAR	2	.2706	.1362	-	.4068
throughput	2.4127				
latency	1.9186				

case 2: total number of nodes = 120					
	# nodes	recv	comp	send	total
Parallel read	8	.0559	-	.1604	.2163
Doppler filter	24	.0254	.1221	.0839	.2313
easy weight	6	.0920	.1110	.0004	.2034
hard weight	56	.0526	.1432	.0045	.2003
easy BF	6	.1003	.0960	.0003	.1966
hard BF	8	.0918	.0928	.0003	.1849
pulse compr	8	.0727	.0999	.0151	.1877
CFAR	4	.1185	.0683	-	.1867
throughput	5.3883				
latency	0.9226				

case 3: total number of nodes = 240					
	# nodes	recv	comp	send	total
Parallel read	16	.1269	-	.0276	.1545
Doppler filter	48	.0833	.0463	.0245	.1541
easy weight	12	.0891	.0558	.0002	.1451
hard weight	112	.0749	.0724	.0004	.1477
easy BF	12	.0975	.0485	.0003	.1463
hard BF	16	.0924	.0516	.0003	.1443
pulse compr	16	.0869	.0502	.0077	.1448
CFAR	8	.1104	.0343	-	.1447
throughput	6.8438				
latency	0.3890				

Table 5. Performance results on the Paragon with the I/O implemented as a separate task.

PFS stripe factor = 64

case 1: total number of nodes = 60					
	# nodes	recv	comp	send	total
Parallel read	4	.0628	-	.3391	.4019
Doppler filter	12	.0085	.2670	.1755	.4510
easy weight	3	.1425	.2217	.0002	.3645
hard weight	28	.0763	.2847	.0003	.3613
easy BF	3	.1621	.1914	.0003	.3537
hard BF	4	.1740	.1759	.0002	.3501
pulse compr	4	.1213	.1980	.0296	.3489
CFAR	2	.2125	.1362	-	.3488
throughput	2.8234				
latency	1.7309				

case 2: total number of nodes = 120					
	# nodes	recv	comp	send	total
Parallel read	8	.0362	-	.1685	.2047
Doppler filter	24	.0280	.1084	.0786	.2151
easy weight	6	.0816	.1111	.0024	.1951
hard weight	56	.0461	.1438	.0003	.1903
easy BF	6	.0914	.0959	.0003	.1877
hard BF	8	.0891	.0908	.0003	.1802
pulse compr	8	.0672	.0999	.0151	.1822
CFAR	4	.1131	.0683	-	.1815
throughput	5.5262				
latency	0.9137				

case 3: total number of nodes = 240					
	# nodes	recv	comp	send	total
Parallel read	16	.0171	-	.0617	.0788
Doppler filter	48	.0073	.0502	.0290	.0864
easy weight	12	.0503	.0558	.0002	.1063
hard weight	112	.0305	.0724	.0029	.1057
easy BF	12	.0491	.0489	.0004	.0984
hard BF	16	.0417	.0540	.0004	.0961
pulse compr	16	.0393	.0502	.0078	.0973
CFAR	8	.0629	.0343	-	.0972
throughput	10.2111				
latency	0.5193				

Table 6. Performance results on the SP with the I/O implemented as a separate task.

PIOFS stripe factor = 80

case 1: total number of nodes = 20 Time in seconds

	# nodes	recv	comp	send	total
Parallel read	2	.1787	-	.1413	.3200
Doppler filter	6	.0045	.0724	.2548	.3316
easy weight	1	.2269	.1047	.0001	.3317
hard weight	7	.1165	.2150	.0013	.3329
easy BF	1	.0641	.0822	.2082	.3545
hard BF	1	.0416	.1179	.1874	.3469
pulse compr	1	.1459	.1538	.0656	.3653
CFAR	1	.2926	.0801	-	.3727
throughput				2.6670	
latency				2.6715	

case 2: total number of nodes = 34 Time in seconds

	# nodes	recv	comp	send	total
Parallel read	4	.1230	-	.0594	.1823
Doppler filter	8	.0264	.0549	.0913	.1726
easy weight	1	.0639	.1043	.0001	.1683
hard weight	14	.0598	.1090	.0003	.1692
easy BF	2	.0576	.0415	.0814	.1805
hard BF	2	.0593	.0596	.0579	.1768
pulse compr	2	.0278	.0784	.0803	.1864
CFAR	1	.1092	.0804	-	.1896
throughput				5.2819	
latency				1.2766	

case 3: total number of nodes = 68 Time in seconds

	# nodes	recv	comp	send	total
Parallel read	8	.1100	-	.0185	.1285
Doppler filter	16	.0455	.0283	.0631	.1369
easy weight	2	.0901	.0535	.0001	.1437
hard weight	28	.0839	.0554	.0001	.1395
easy BF	4	.1158	.0208	.0035	.1401
hard BF	4	.0813	.0483	.0089	.1385
pulse compr	4	.1008	.0391	.0054	.1453
CFAR	2	.1074	.0404	-	.1478
throughput				6.5063	
latency				0.6531	

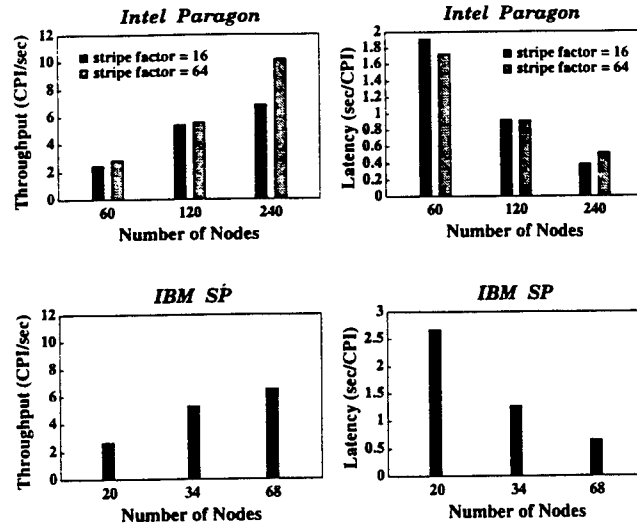


Figure 9. Performance results for the implementation using a separate I/O task. This figure corresponds to Tables 4, 5, and 6.

same number of compute nodes assigned and hence have approximately the same computation time. Therefore, the execution times of these tasks have no significant differences for both cases and the throughput results do not change significantly.

The latency, on the other hand, is the sum of the execution times of all the tasks except for the tasks with temporal data dependency, that is, easy and hard weight computation tasks (T_2 and T_3 , respectively.) In the design with a separate I/O task, the latency contains one more term than the embedded I/O implementation: the execution time of the new task, T_0 . Therefore, the latency results become worse in this implementation.

7 Task Combination

From the comparison of performance results for the two I/O task implementations, we notice that the structure of the STAP pipeline system can be reorganized to improve the latency. The first implementation that embeds I/O in the Doppler filter processing task can

be viewed as combining the first two tasks of the second implementation that uses a separate task for I/O. As shown in Section 6.2, the first I/O implementation has a better latency performance, while the throughput results are approximately the same.

7.1 Improving latency

We investigate whether the latency can be further improved by combining multiple tasks of the pipeline into a single task. We consider Tables 1, 2, and 3 as an example and combine the last two tasks, the pulse compression and CFAR processing tasks, into a single task. In order to make a fair comparison, we keep the total number of nodes allocated to the whole pipeline system to be the same. The number of nodes assigned to this single task is equal to the sum of the nodes assigned to the two tasks in the original pipeline. In this case, no communication costs between pulse compression and CFAR processing tasks are incurred. Tables 7, 8, and 9 give the timing results corresponding to Tables 1, 2, and 3 with the same total number of nodes assigned to the pipeline system. Figure 10 shows the bar charts of the throughput and latency results for Tables 7, 8, and 9. Figure 11 gives a comparison of performance results of the STAP pipeline system with and without task combining. We observe that the latency improves for all cases on both Paragon PFS and SP PIOFS file systems when the last two tasks are combined.

This improvement can also be explained by examining the latency equation. Before task combination, the latency equation for the STAP pipeline system with 7 tasks is

$$\text{latency}_7 = T_0 + \max(T_3, T_4) + T_5 + T_6. \quad (5)$$

Let W_5 and W_6 be the workloads for tasks 5 and 6, respectively. The execution times for task 5 and 6 are

$$T_5 = \frac{W_5}{P_5} + C_5 + V_5 \quad (6)$$

and

$$T_6 = \frac{W_6}{P_6} + C_6 + V_6 \quad (7)$$

where C_i and V_i represent the communication time and the other parallelization overhead for task i respectively. Similarly, let T_{5+6} be the execution time of the task that combines tasks 5 and 6 running on $P_5 + P_6$ nodes:

$$T_{5+6} = \frac{W_5 + W_6}{P_5 + P_6} + C_{5+6} + V_{5+6}. \quad (8)$$

By subtracting Equations (6) and (7) from Equation (8), we have

$$\begin{aligned} T_{5+6} - (T_5 + T_6) &= \frac{W_5 + W_6}{P_5 + P_6} - \frac{W_5}{P_5} - \frac{W_6}{P_6} \\ &+ C_{5+6} - C_5 - C_6 \\ &+ V_{5+6} - V_5 - V_6 \end{aligned} \quad (9)$$

where

$$\begin{aligned} &\frac{W_5 + W_6}{P_5 + P_6} - \frac{W_5}{P_5} - \frac{W_6}{P_6} \\ &= \frac{-W_5 P_6^2 - W_6 P_5^2}{P_5 P_6 (P_5 + P_6)} \\ &< 0. \end{aligned} \quad (10)$$

Communication for the combined task occurs only when receiving data from tasks 3 and 4. Prior to the task combination, the same communication takes place in the receive phase of task 5. The difference is the number of nodes used between the two tasks. Since $P_{5+6} > P_5$, the data size for each received message from tasks 3 and 4 to the combined task is smaller than that for task 5. Besides, in task 5, C_5 includes the communication cost of sending messages from task 5 to task 6 which does not occur in the combined task. Hence, we have

$$C_{5+6} < C_5. \quad (11)$$

The remaining overhead, V_i , is due to parallelization of task i . Since the operations in

Table 7. Performance results on the Paragon with pulse compression and CFAR tasks combined.

PFS stripe factor = 16

case 1: total number of nodes = 56					
	# nodes	recv	comp	send	total
Doppler filter	12	.0094	.2589	.0908	.3591
easy weight	3	.1307	.2230	.0002	.3540
hard weight	28	.0660	.2868	.0003	.3531
easy BF	3	.1449	.1930	.0003	.3382
hard BF	4	.1616	.1756	.0003	.3375
PC + CFAR	6	.1517	.1863	-	.3380
throughput				2.9243	
latency				0.7913	

case 2: total number of nodes = 112					
	# nodes	recv	comp	send	total
Doppler filter	24	.0194	.1294	.0656	.2145
easy weight	6	.0831	.1111	.0002	.1944
hard weight	56	.0468	.1427	.0046	.1940
easy BF	6	.0914	.0958	.0003	.1874
hard BF	8	.0892	.0887	.0004	.1784
PC + CFAR	12	.0869	.0935	-	.1804
throughput				5.5340	
latency				0.4221	

case 3: total number of nodes = 224					
	# nodes	recv	comp	send	total
Doppler filter	48	.0953	.0623	.0323	.1900
easy weight	12	.1056	.0558	.0003	.1617
hard weight	112	.0930	.0726	.0004	.1661
easy BF	12	.1116	.0484	.0003	.1603
hard BF	16	.1063	.0513	.0004	.1579
PC + CFAR	24	.1079	.0513	-	.1592
throughput				6.1478	
latency				0.2948	

Table 8. Performance results on the Paragon with pulse compression and CFAR tasks combined.

PFS stripe factor = 64

case 1: total number of nodes = 56					
	# nodes	recv	comp	send	total
Doppler filter	12	.0319	.2485	.0915	.3718
easy weight	3	.1265	.2218	.0002	.3485
hard weight	28	.0631	.2839	.0003	.3473
easy BF	3	.1400	.1921	.0003	.3324
hard BF	4	.1533	.1756	.0003	.3292
PC + CFAR	6	.1449	.1860	-	.3309
throughput				3.0027	
latency				0.7957	

case 2: total number of nodes = 112					
	# nodes	recv	comp	send	total
Doppler filter	24	.0104	.1301	.0528	.1933
easy weight	6	.0774	.1111	.0002	.1887
hard weight	56	.0438	.1427	.0022	.1886
easy BF	6	.0853	.0959	.0003	.1815
hard BF	8	.0869	.0886	.0004	.1759
PC + CFAR	12	.0838	.0936	-	.1773
throughput				5.6029	
latency				0.4197	

case 3: total number of nodes = 224					
	# nodes	recv	comp	send	total
Doppler filter	48	.0071	.0676	.0306	.1054
easy weight	12	.0522	.0559	.0002	.1083
hard weight	112	.0347	.0730	.0031	.1108
easy BF	12	.0533	.0482	.0004	.1018
hard BF	16	.0481	.0512	.0003	.0997
PC + CFAR	24	.0489	.0514	-	.1003
throughput				9.8853	
latency				0.2392	

Table 9. Performance results on the SP with pulse compression and CFAR tasks combined.

PIOFS stripe factor = 80

case 1: total number of nodes = 18					
	# nodes	recv	comp	send	total
Doppler filter	6	.1320	.0728	.1894	.3942
easy weight	1	.2844	.1023	.0001	.3868
hard weight	7	.1738	.2131	.0002	.3870
easy BF	1	.3039	.0823	.0001	.3862
hard BF	1	.2677	.1182	.0002	.3862
PC + CFAR	2	.2683	.1194	-	.3877
throughput	2.5754				
latency	0.9388				

case 2: total number of nodes = 30					
	# nodes	recv	comp	send	total
Doppler filter	8	.1105	.0550	.1055	.2710
easy weight	1	.1711	.1026	.0002	.2739
hard weight	14	.1570	.1077	.0002	.2649
easy BF	2	.2225	.0417	.0001	.2644
hard BF	2	.2051	.0608	.0002	.2661
PC + CFAR	3	.1878	.0793	-	.2671
throughput	3.7492				
latency	0.6255				

case 3: total number of nodes = 60					
	# nodes	recv	comp	send	total
Doppler filter	16	.1044	.0279	.0462	.1786
easy weight	2	.1350	.0515	.0002	.1867
hard weight	28	.1238	.0568	.0002	.1808
easy BF	4	.1582	.0210	.0002	.1794
hard BF	4	.1485	.0300	.0003	.1787
PC + CFAR	6	.1397	.0414	-	.1810
throughput	5.5356				
latency	0.4207				

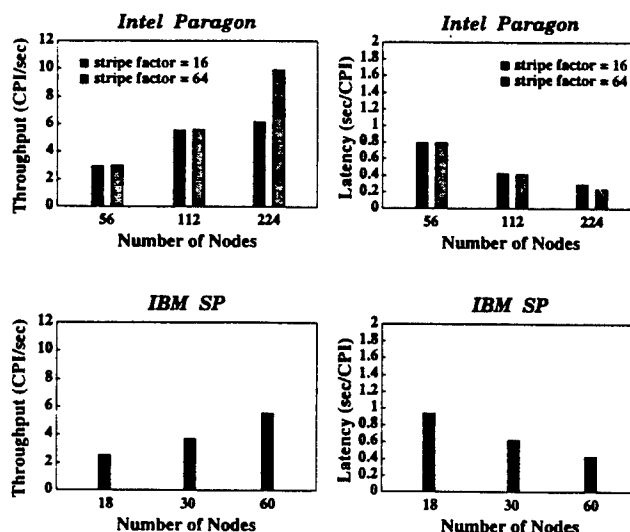


Figure 10. Performance results for the STAP pipeline system that combines the pulse compression and CFAR tasks into a single task. This figure corresponds to Tables 7, 8, and 9.

tasks 5 and 6 are sets of individual subroutines which require no communication within each single task, parallelization is carried out by evenly partitioning these subroutines among the nodes assigned. Due to this computational structure, the overhead for these two tasks becomes negligible compared to their communication costs. From Equations (9), (10), and (11) we can conclude that

$$T_{5+6} < T_5 + T_6. \quad (12)$$

Therefore, the new latency equation of the STAP pipeline system with the last two tasks combined becomes

$$\begin{aligned} \text{latency}_6 &= T_0 + \max(T_3, T_4) + T_{5+6} \\ &< \text{latency}_7 \end{aligned} \quad (13)$$

Combining the last two tasks, therefore, reduces the latency.

Table 10 gives the percentage of improvement in latency when the last two tasks are combined. These improvements were made without adding any extra nodes to the pipeline

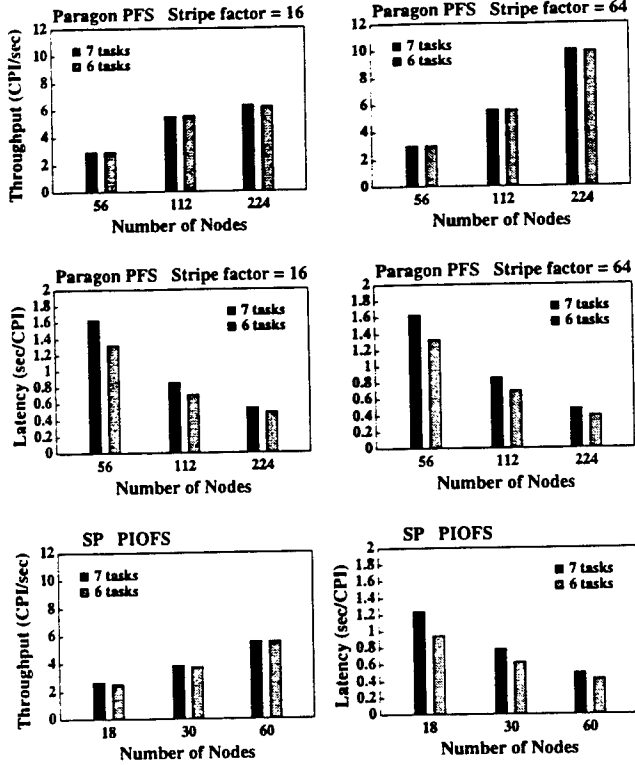


Figure 11. Performance comparison of the pipeline system with and without task combining. The throughput results remain approximately the same. Latency is improved when the last two tasks are combined.

system. We observe that the percentage decreases as the number of nodes goes up. Normally, scalability of the parallelization tends to decrease when more processors are used. This also explains the trend for the percentage improvement shown in Table 10. Notice that the tasks that can be combined to improve the latency do not include tasks with temporal data dependency. It is because only those tasks with spatial data dependency contribute to the latency.

7.2 Improving throughput

The throughput results, on the other hand, do not change significantly when the two tasks are combined. This is because the throughput

Table 10. Percentage of latency improvement when the Pulse compression and CFAR tasks are combined into a single task.

Paragon: PFS

# nodes	56	112	224
16 stripe dir	19.3%	18.4%	10.4%
64 stripe dir	18.7%	18.4%	16.7%

SP: PIOFS

# nodes	18	30	60
80 stripe dir	24.0%	19.9%	15.9%

is determined by the task with the maximum execution time among all the tasks, which is still the maximum in the new pipeline system. Assuming that T_{max} is the maximum execution time before task combination, the throughput is given by

$$throughput_7 = \frac{1}{T_{max}}$$

where

$$T_{max} = \max_{0 \leq i < 7} T_i \\ \geq \max(T_5, T_6)$$

From Equations (6), (7), and (8), the execution time of the new combined task becomes

$$\begin{aligned} T_{5+6} &\approx \frac{P_5 T_5 + P_6 T_6}{P_5 + P_6} \\ &\leq \frac{P_5 \max(T_5, T_6) + P_6 \max(T_5, T_6)}{P_5 + P_6} \\ &= \max(T_5, T_6) \end{aligned} \quad (14)$$

and the new maximum execution time becomes

$$\begin{aligned} T'_{max} &= \max(T_0, T_1, T_2, T_3, T_4, T_{5+6}) \\ &\leq \max(T_0, T_1, T_2, T_3, T_4, T_5, T_6) \\ &= T_{max}. \end{aligned}$$

Therefore, the throughput will not decrease after task combination because

$$\begin{aligned} \text{throughput}_6 &= \frac{1}{T'_{max}} \\ &\geq \frac{1}{T_{max}} \\ &= \text{throughput}_7. \end{aligned} \quad (15)$$

Both latency and throughput can be improved simultaneously when one of the combined tasks determines the throughput of the pipeline system. Suppose that either task 5 or task 6 has the maximum execution time among all the 7 tasks in the STAP pipeline system, that is,

$$\begin{aligned} T_{max} &= \max(T_5, T_6) \\ &> \max_{0 \leq i \leq 4} T_i. \end{aligned} \quad (16)$$

Notice that none of these two tasks has temporal data dependency. From Equation (13), we have latency improvement when tasks 5 and 6 are combined. From Equations (15) and (16), the throughput is increased. The reduction of execution time of both tasks 5 and 6 contributes to the latency as well as to the throughput. Therefore, not only the throughput can be increased, but the latency can be also reduced. Note that in our experiment results shown in the previous section, the task with the maximum execution time is neither task 5 nor task 6, that is, $T_{max} > \max(T_5, T_6)$.

8 Raw CPI data redistribution

The presentation in this paper up to now assumed that a special hardware is available to pre-process the raw CPI data received by the radar before delivering it to the STAP pipeline system. However, this special purpose equipment may not perform very efficiently or may not be available. We investigate the possibility of implementing this data pre-processing operation using a software approach. Actually,

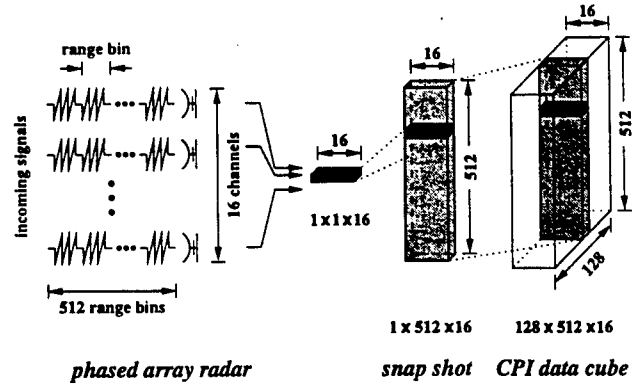


Figure 12. Raw CPI data received from a phased array radar is used to form a $128 \times 512 \times 16$ three dimensional data cube.

Air Force Research Laboratory (AFRL) performed a real time STAP demonstration using exactly the same signal processing algorithm as ours onboard an airborne platform in May 1996 [11, 10]. The radar was a phased array L-Band radar with 32 elements organized into two rows of 16 each. Only the data from the upper 16 elements were processed with STAP. This data is a 1.25 MHz intermediate frequency (IF) signal that is 4:1 oversampled at 5 MHz. The number representation at IF is 14 bits, 2's complement and is converted to 16 bit baseband real and imaginary numbers. Special interface boards were used to digitally demodulate to baseband. The signal data formed a raw 3-dimensional data cube called coherent processing interval (CPI) data cube comprised of 128 pulses, 512 range gates (32.8 miles), and 16 channels, shown in Figure 12. These special interface boards were also used to corner turn the data cube so that CPI is unit stride along pulses. It speeds the subsequent Doppler processing on the High Performance Computing (HPC) systems. Live CPI data from a phased-array radar were processed by a ruggedized version of the Paragon computer. The STAP algorithm was performed on this computer using the raw data from the 16 columns of the phased array.

All experiments described in the previous sections assumed that this special purpose hardware was used to pre-process the raw CPI data such that each CPI data cube is corner-turned from $128 \times 512 \times 16$ to $512 \times 16 \times 128$ and each complex element in a CPI is type-converted from two 16-bits real numbers to two 32-bits real numbers (type float in C language.) The operations of corner turn and CPI data partitioning among compute nodes are illustrated in Figure 13. The reason for the corner turn operations is that the major operations in the Doppler filter processing task, the Fast Fourier transforms (FFTs), need to be performed along the pulse dimension of the CPI cube. That is, 128-point FFTs are performed for every range and channel. The corner turn operation, here, is to allow each FFT to be computed on a single compute node in the Doppler filter processing task. Given this hardware, the parallel pipeline STAP system can directly process the CPI data without redistributing it among the compute nodes once the CPI data is read from the disk.

8.1 Corner turn and type conversion

Without hardware support for the operations of corner turn and type conversion, the parallel pipeline STAP system has to include this in its implementation. In order that every FFT can be processed in a single compute node in the Doppler filter processing task, the CPI data has to be partitioned along the dimension of range cells among the compute nodes assigned, shown in Figure 13(d). Note that two consecutive pulses in a raw CPI data cube are stored in disks at a distance of $512 \cdot 16$ complex numbers. By partitioning the raw CPI along the range dimension, each sub-CPI data for one node consists of several pieces of non-contiguous data. For instance, we use 4 nodes to read a raw CPI data cube and it results in a sub-CPI of size $128 \times 128 \times 16$. That is, each

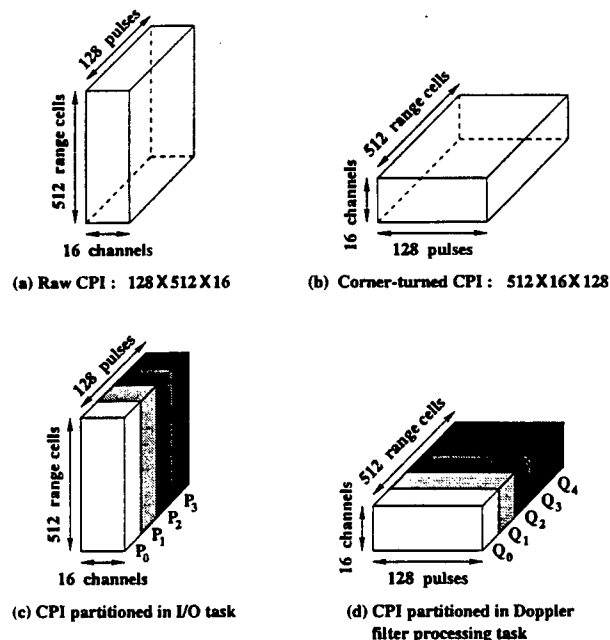


Figure 13. (a) Raw CPI data received from the radar as a $128 \times 512 \times 16$ data cube. (b) Corned-turned CPI data cube of size $512 \times 16 \times 128$. (c) Raw CPI partitioned among 4 reading nodes. (d) Corned-turned CPI partitioned among 5 nodes.

sub-CPI has 128 pieces of data and each piece is of size 128×16 . Although contents of each data piece are stored contiguously in disks, the 128 data pieces themselves are not adjacent to each other. To obtain the sub-CPI data required by each node, two implementations for reading CPI data can be done:

1. Every node performs several read operations directly from the disks. Each read is for a data piece of a sub-CPI. After the sub-CPI data is read, type-conversion operations are applied.
2. Using a two-phase I/O access strategy [5], the CPI data is first read using data distribution which conforms with the distribution of CPI data over the disks. This results in each node making a single, large, and contiguous disk space access. In

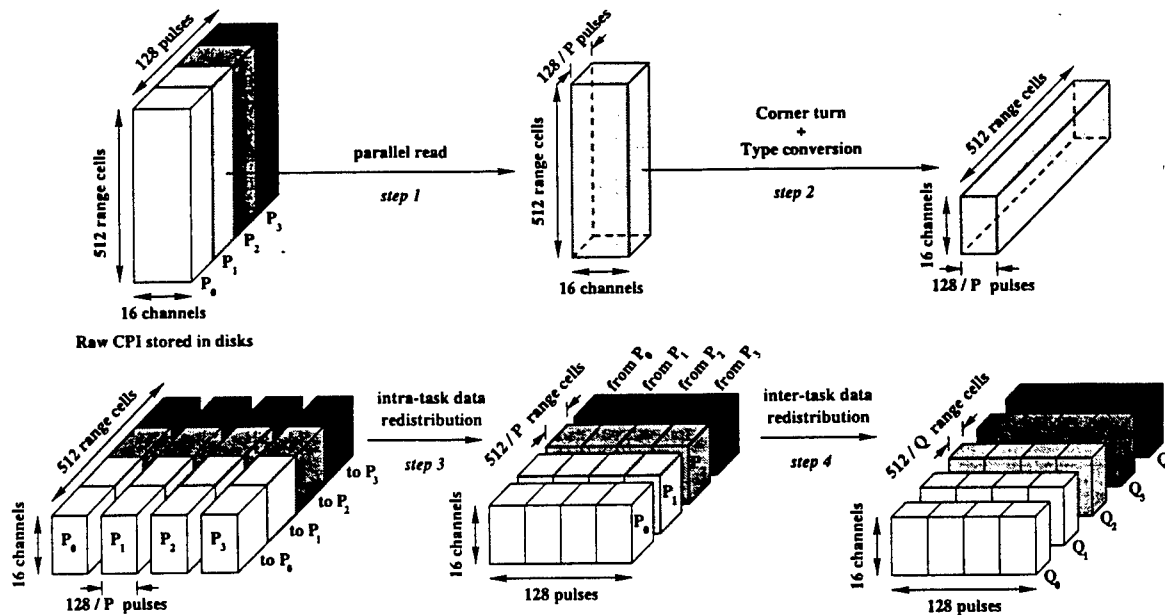


Figure 14. Implementation of parallel reading of raw CPI data from disks and its distribution for the Doppler filter processing task.

the second phase, the sub-CPI data is type-converted, corner-turned, and redistributed among the nodes to match the desired data distribution.

Two-phase I/O access strategy has been shown to improve the I/O performance significantly. This method first reduces the I/O bottleneck from disks to compute nodes by making all the file accesses large and contiguous. Second, the data redistribution uses the inter-processor communication network with higher bandwidth and higher degree of connectivity.

8.2 Implementation

To read CPI files in parallel, we implemented the two-phase I/O access strategy on the two STAP pipeline system I/O designs described in Section 5. The implementation for the reading of CPI files for the STAP pipeline system with a separate I/O task is shown in Figure 14. In this implementation, each node in the I/O task performs the following steps:

1. uses one read operation to read an exclusive part of CPI data. In other words, the CPI data is partitioned into exclusive subsets and node i in the I/O task reads the i^{th} subset of each CPI file.
2. performs the corner turn and type conversion operations on the sub-CPI data.
3. redistributes the sub-CPI data with other nodes in the I/O task such that each node receives all parts of sub-CPI data it is responsible for. Data exchange in this step is an all-to-all personalized communication within the same group of nodes.
4. sends the re-organized sub-CPI data to the Doppler filter processing task. The communication pattern in this step is a left-right shift communication. Notice that the number of nodes assigned to the I/O task may be different from the Doppler filter processing task.

Table 11. Performance results on the Paragon with the I/O implemented as a separate task in which the corner turn and type conversion are embedded in the receive phase.

PFS stripe factor = 16

case 1: total number of nodes = 64					Time in seconds
	# nodes	recv	comp	send	total
Parallel read	8	.3256	-	.0003	.3259
Doppler filter	12	.0634	.1744	.0907	.3285
easy weight	3	.1053	.2215	.0002	.3270
hard weight	28	.0403	.2849	.0003	.3255
easy BF	3	.1204	.1923	.0003	.3131
hard BF	4	.1346	.1757	.0003	.3105
pulse compr	4	.0812	.1978	.0302	.3092
CFAR	2	.1726	.1361	-	.3087
throughput	3.2079				
latency	1.2516				

case 2: total number of nodes = 128					Time in seconds
	# nodes	recv	comp	send	total
Parallel read	16	.1485	-	.0099	.1585
Doppler filter	24	.0037	.0976	.0580	.1593
easy weight	6	.0528	.1110	.0002	.1639
hard weight	56	.0161	.1435	.0038	.1634
easy BF	6	.0515	.0969	.0004	.1488
hard BF	8	.0555	.0894	.0003	.1452
pulse compr	8	.0313	.1000	.0151	.1464
CFAR	4	.0777	.0682	-	.1459
throughput	6.7809				
latency	0.7797				

case 3: total number of nodes = 256					Time in seconds
	# nodes	recv	comp	send	total
Parallel read	32	.1041	-	.0004	.1045
Doppler filter	48	.0241	.0453	.0244	.0937
easy weight	12	.0499	.0559	.0002	.1060
hard weight	112	.0319	.0729	.0008	.1056
easy BF	12	.0516	.0486	.0003	.1005
hard BF	16	.0474	.0518	.0003	.0996
pulse compr	16	.0411	.0499	.0079	.0989
CFAR	8	.0643	.0343	-	.0986
throughput	9.9740				
latency	0.3713				

Table 12. Performance results on the Paragon with the I/O implemented as a separate task in which the corner turn and type conversion are embedded in the receive phase.

PFS stripe factor = 64

case 1: total number of nodes = 64					Time in seconds
	# nodes	recv	comp	send	total
Parallel read	8	.3242	-	.0004	.3246
Doppler filter	12	.0575	.1742	.0956	.3272
easy weight	3	.1039	.2214	.0002	.3255
hard weight	28	.0375	.2849	.0003	.3227
easy BF	3	.1197	.1921	.0003	.3121
hard BF	4	.1275	.1830	.0002	.3108
pulse compr	4	.0789	.1980	.0296	.3065
CFAR	2	.1693	.1360	-	.3053
throughput	3.3022				
latency	1.2889				

case 2: total number of nodes = 128					Time in seconds
	# nodes	recv	comp	send	total
Parallel read	16	.1471	-	.0163	.1633
Doppler filter	24	.0048	.1004	.0669	.1722
easy weight	6	.0601	.1109	.0002	.1712
hard weight	56	.0214	.1430	.0059	.1703
easy BF	6	.0524	.0970	.0003	.1497
hard BF	8	.0605	.0895	.0003	.1503
pulse compr	8	.0369	.0994	.0149	.1512
CFAR	4	.0825	.0681	-	.1506
throughput	6.5610				
latency	0.8300				

case 3: total number of nodes = 256					Time in seconds
	# nodes	recv	comp	send	total
Parallel read	32	.0908	-	.0005	.0913
Doppler filter	48	.0015	.0507	.0244	.0766
easy weight	12	.0434	.0559	.0002	.0995
hard weight	112	.0248	.0727	.0005	.0980
easy BF	12	.0455	.0499	.0003	.0957
hard BF	16	.0390	.0548	.0004	.0942
pulse compr	16	.0349	.0505	.0078	.0932
CFAR	8	.0590	.0342	-	.0932
throughput	10.5710				
latency	0.4629				

Table 13. Performance results on the Paragon with the I/O implemented in the Doppler filter processing task in which the corner turn and type conversion are embedded in the receive phase.

PFS stripe factor = 16

case 1: total number of nodes = 31					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	8	.3188	.2584	.1354	.7127
easy weight	2	.3794	.3321	.0002	.7118
hard weight	14	.1446	.5669	.0004	.7119
easy BF	2	.4164	.2865	.0002	.7031
hard BF	2	.3405	.3478	.0002	.6886
pulse compr	2	.2313	.3949	.0583	.6845
CFAR	1	.4121	.2724	-	.6845
throughput					1.4411
latency					1.9326

case 2: total number of nodes = 60					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	16	.1505	.1296	.0681	.3482
easy weight	3	.1277	.2216	.0002	.3495
hard weight	28	.0629	.2849	.0003	.3481
easy BF	3	.1419	.1918	.0003	.3340
hard BF	4	.1537	.1756	.0002	.3295
pulse compr	4	.1003	.1985	.0298	.3286
CFAR	2	.1918	.1363	-	.3281
throughput					3.0129
latency					0.9789

case 3: total number of nodes = 120					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	32	.0863	.0660	.0349	.1872
easy weight	6	.0780	.1110	.0002	.1893
hard weight	56	.0431	.1429	.0019	.1879
easy BF	6	.0842	.0961	.0003	.1806
hard BF	8	.0886	.0880	.0003	.1770
pulse compr	8	.0616	.0995	.0151	.1763
CFAR	4	.1079	.0683	-	.1762
throughput					5.5923
latency					0.5047

case 4: total number of nodes = 238					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	64	.0625	.0364	.0192	.1181
easy weight	12	.0675	.0557	.0003	.1234
hard weight	112	.0494	.0721	.0004	.1219
easy BF	12	.0732	.0482	.0004	.1218
hard BF	14	.0649	.0511	.0003	.1164
pulse compr	16	.0587	.0501	.0078	.1166
CFAR	8	.0821	.0344	-	.1165
throughput					8.4272
latency					0.2925

Table 14. Performance results on the Paragon with the I/O implemented in the Doppler filter processing task in which corner turn and type conversion are embedded in the receive phase.

PFS stripe factor = 64

case 1: total number of nodes = 31					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	8	.3196	.2586	.1355	.7138
easy weight	2	.3804	.3321	.0003	.7128
hard weight	14	.1455	.5670	.0004	.7129
easy BF	2	.4174	.2865	.0002	.7042
hard BF	2	.3413	.3480	.0003	.6896
pulse compr	2	.2321	.3949	.0582	.6852
CFAR	1	.4129	.2724	-	.6852
throughput					1.4390
latency					1.9368

case 2: total number of nodes = 60					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	16	.1504	.1298	.0757	.3558
easy weight	3	.1341	.2216	.0002	.3559
hard weight	28	.0697	.2849	.0004	.3550
easy BF	3	.1486	.1913	.0003	.3402
hard BF	4	.1524	.1828	.0002	.3355
pulse compr	4	.1007	.1989	.0317	.3313
CFAR	2	.1918	.1363	-	.3280
throughput					3.0618
latency					1.0159

case 3: total number of nodes = 120					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	32	.0835	.0647	.0455	.1937
easy weight	6	.0872	.1111	.0002	.1985
hard weight	56	.0469	.1430	.0074	.1973
easy BF	6	.0934	.0959	.0003	.1896
hard BF	8	.0864	.0895	.0003	.1762
pulse compr	8	.0618	.0998	.0151	.1768
CFAR	4	.1080	.0683	-	.1763
throughput					5.6552
latency					0.5264

case 4: total number of nodes = 238					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	64	.0617	.0327	.0190	.1134
easy weight	12	.0675	.0558	.0002	.1236
hard weight	112	.0497	.0724	.0004	.1225
easy BF	12	.0735	.0482	.0003	.1220
hard BF	14	.0652	.0511	.0003	.1166
pulse compr	16	.0590	.0500	.0077	.1167
CFAR	8	.0824	.0343	-	.1167
throughput					8.4237
latency					0.2927

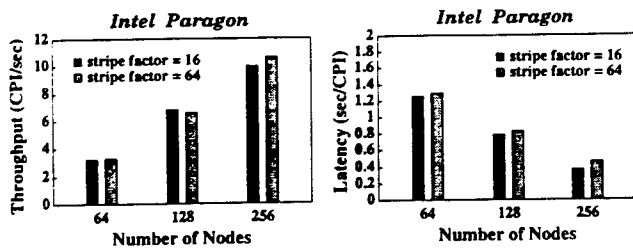


Figure 15. Performance results for the implementation using a separate I/O task in which the corner turn and type conversion are embedded in the receive phase. This figure corresponds to Tables 11 and 12.

In the first I/O design that embeds the I/O in the Doppler filter processing task, the only difference is that it is without step 4, the left-right shift communication. In addition, all the steps are performed within the same group of nodes. The sub-CPI data redistribution is performed within the same group of compute nodes in the Doppler filter processing task. As opposed to the inter-task data dependency discussed in Section 2, this data redistribution results in an intra-task data dependency. The intra-task dependency exists when intermediate results need to be exchanged during the execution of a single parallel task in the pipeline.

8.3 Performance results

The performance results for the implementation using a separate I/O task are given in Tables 11 and 12, for Paragon PFS file systems with 16 and 64 striped directories, respectively. Figure 15 shows the bar charts corresponding to Tables 11 and 12. Linear speedups were obtained for both throughput and latency.

The performance results for the implementation with the I/O task embedded in the Doppler filter processing task is shown in Tables 13 and 14, for Paragon PFS file systems with 16 and 64 striped directories, respectively. Figure 16 shows the bar charts corresponding to Tables 13 and 14. We observe that the

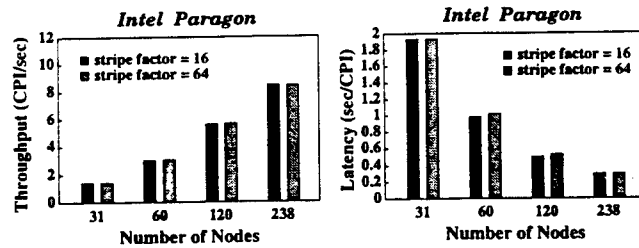


Figure 16. Performance results for the implementation when the parallel I/O, corner turn, and type conversion are embedded in the receive phase of the Doppler filter processing task. This figure corresponds to Tables 13 and 14.

throughput and latency show linear speedups till the case with a total of 120 nodes. The timing for performing read CPI data from disks, corner turn, type conversion, and CPI data redistribution are included in the receive phase of the Doppler filter processing task. When we increase the number of nodes from 32 to 64 in the Doppler filter processing task, the performance of the receive phase does not scale up linearly. This is because of the increasing cost of the all-to-all personalized communication in the sub-CPI data redistribution. The size of each CPI data in our experiments is $128 \cdot 512 \cdot 16 \cdot (2 \cdot 4 \text{ bytes}) = 8\text{M bytes}$. With 64 nodes, the size of data in each send/receive of the all-to-all personalized communication becomes $\frac{8\text{M bytes}}{64 \cdot 64} = 2\text{K bytes}$. In the all-to-all personalized communication, each node has a total of 64 read/receive calls whose communication startup time overwhelms the message transmission time with respect to the relatively small size of the messages (2K bytes each.)

9 Conclusions

In this work, we studied the effects of parallel I/O implementation on the parallel pipeline system for a modified PRI-staggered post-Doppler STAP algorithm. The parallel

pipeline STAP system was run portably on Intel Paragon and IBM SP and the overall performance results demonstrated the linear scalability of our parallel pipeline design when the existing parallel file systems were used in the I/O implementations. On the Paragon, we found that a pipeline bottleneck can result when using a parallel file system with a relatively smaller stripe factor. With a larger stripe factor, a parallel file system can deliver higher efficiency of I/O operations and, therefore, improve the throughput performance.

This paper presented two I/O designs which are incorporated into the parallel pipeline STAP system. One embedded I/O in the original pipeline and the other used a separate I/O task. By comparing the results of these designs, we found that the task structure of the pipeline can be reorganized to further improve the latency. Without adding any compute nodes, we obtained performance improvement in the latency when the last two tasks were combined. We also analyzed the possibility of further improvement by examining the throughput and latency equations.

We also investigated a software approach to implement raw data pre-processing which can often be done by a special purpose hardware. The performance results demonstrate that the parallel pipeline STAP system scaled well even with a more complicated I/O implementation.

10 Acknowledgments

This work was supported by Air Force Materials Command under contract F30602-97-C-0026. We acknowledge the use of the Intel Paragon at California Institute of Technology and the IBM SP at Argonne National Laboratory.

References

- [1] R. Brown and R. Linderman. Algorithm Development for an Airborne Real-Time STAP Demonstration. In *Proceedings of the IEEE National Radar Conference*, 1997.
- [2] A. Choudhary, W. Liao, D. Weiner, P. Varshney, R. Linderman, and M. Linderman. Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers. *International Parallel Processing Symposium*, 1998.
- [3] A. Choudhary and J. Patel. *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems*. Kluwer Academic Publishers, Boston, MA, 1990.
- [4] A. Choudhary and R. Ponnusamy. Parallel Implementation and Evaluation of a Motion Estimation System Algorithm using Several Data Decomposition Strategies. *Journal of Parallel and Distributed Computing*, 14:50-65, January 1992.
- [5] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS'93*, pages 56-70, 1993.
- [6] IBM Corp. *IBM AIX Parallel I/O File System: Installation, Administration, and Use*, October 1996.
- [7] Intel Corporation. *Paragon System User's Guide*, Apr. 1996.
- [8] W. Liao, A. Choudhary, D. Weiner, and P. Varshney. Multi-Threaded Design and Implementation of Parallel Pipelined STAP on Parallel Computers with SMP Nodes. *International Parallel Processing Symposium*, 1999.
- [9] M. Linderman and R. Linderman. Real-Time STAP Demonstration on an Embedded High Performance Computer. In *Proceedings of the IEEE National Radar Conference*, 1997.
- [10] M. Linderman and R. Linderman. Real-Time STAP Demonstration on an Embedded High Performance Computer. *IEEE AES Systems Magazine*, pages 15-21, Mar. 1998.
- [11] M. Little and W. Berry. Real-Time Multi-Channel Airborne Radar Measurements. In *Proceedings of the IEEE National Radar Conference*, 1997.

Appendix C

Ph.D. Dissertation

ABSTRACT OF DISSERTATION

This dissertation presents a parallel pipelined computational model for radar signal processing applications. Performance results for the design and implementation of a real Space-Time Adaptive Processing (STAP) application on parallel computers are presented. The dissertation also discusses the process of software development for such an application on parallel computers when latency and throughput are both considered together and presents tradeoffs considered with respect to inter and intra-task communication and data redistribution. The results show that not only scalable performance was achieved for individual component tasks of STAP but linear speedups were obtained for the integrated task performance, both for latency as well as throughput. Multi-threaded design for the STAP application is also presented for the parallel machine with SMP nodes. It is shown that the performance is enhanced when a multi-threaded implementation is employed. In this dissertation, we also study the effect on system performance when the I/O task is incorporated in the parallel pipeline computational model. There are two alternatives for I/O implementation: embedding I/O in the pipeline or having a separate I/O task. From these two I/O implementations, we discovered that the latency may be improved when the structure of the pipeline is reorganized by merging multiple tasks into a single task. All the performance results shown in this work demonstrated the scalability of the parallel pipeline STAP system.

PARALLEL PIPELINED COMPUTATIONAL MODEL
FOR
SPACE-TIME ADAPTIVE PROCESSING

by

WEI-KENG LIAO

Bachelor of Science, National Chung-Hsing University, Taiwan, 1988

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science
in the Graduate School of Syracuse University

June, 1999

Approved _____

Professor Alok N. Choudhary

Date _____

Approved _____

Professor Pramod K. Varshney

Date _____

Approved _____

Professor Donald D. Weiner

Date _____

© Copyright 1999 WEI-KENG LIAO
All Rights Reserved

Contents

List of Tables	viii
List of Figures	xi
Acknowledgments	xv
1 Introduction	1
1.1 Parallelism in STAP	2
1.2 Shared Memory versus Distributed Memory HPC Systems	5
1.3 Disk I/O	7
1.4 Processor Assignment	7
1.5 Organization of the Dissertation	8
2 Parallel Pipeline System Model	10
2.1 Custom VLSI versus HPC Systems	11
2.2 Overview of Space-Time Adaptive Processing	15
2.2.1 STAP Algorithms	16
2.2.2 Computational Characteristic Analysis	18
2.3 Related Work	21
2.4 Parallel Pipeline Computational Model	23
2.4.1 Data Dependencies	25
2.5 Parallelization Issues and Approaches	26
2.5.1 Throughput and Latency	27

2.5.2	Data Redistribution	29
2.5.3	Task Scheduling and Processor Assignment	31
2.6	Summary	32
3	Parallel Pipelined STAP System	34
3.1	Algorithm Overview	36
3.2	Design and Implementation	39
3.2.1	Doppler Filter Processing	41
3.2.2	Weight Computation	43
3.2.3	Beamforming	45
3.2.4	Pulse Compression	48
3.2.5	CFAR Processing	49
3.3	Software Development and System Platform	49
3.4	Performance Results	53
3.4.1	Computation Costs	54
3.4.2	Inter-task Communication	54
3.4.3	Integrated System Performance	58
3.5	Summary	63
4	Multi-Threaded Design and Implementation	65
4.1	Symmetrical Multi-Processor System	66
4.1.1	Intel Paragon MP System	68
4.2	Design and Implementation	69
4.2.1	Threads in the Compute Phase	69
4.2.2	Software Development	71
4.3	Performance Results	71
4.3.1	Compute Time	72
4.3.2	Integrated System Performance Evaluation	74
4.3.3	Tradeoff Between Throughput and Latency	78
4.4	Summary	82

5	I/O Implementation	84
5.1	Parallel File Systems	86
5.1.1	Intel Paragon PFS File System	86
5.1.2	IBM SP PIOFS File System	88
5.2	Design and Implementation	88
5.2.1	I/O Task Implementation	90
5.3	Performance Results	91
5.3.1	I/O Embedded in the First Task	91
5.3.2	A New I/O Task	96
5.4	Task Combination	102
5.4.1	Improving Latency	102
5.4.2	Improving Throughput	110
5.5	Raw CPI Data Redistribution	111
5.5.1	Corner Turn and Type Conversion	113
5.5.2	Implementation	115
5.5.3	Performance Results	116
5.6	Summary	122
6	Summary and Conclusions	124
6.1	Suggestions for Future Work	127

List of Tables

1	Characteristics of shared-memory and distributed-memory HPC systems.	6
2	Feature comparison between custom VLSI and HPC systems.	14
3	Configurations of the system platforms on which we ran the parallel pipeline STAP codes.	51
4	The number of floating point operations for the PRI-staggered post Doppler STAP algorithm to process one CPI data.	53
5	Timing results of inter-task communication. Time in seconds. # proc: number of processors.	56
6	Performance results on the Intel Paragon for 3 cases of processor assignments. Time in seconds. # proc: number of processors.	59
7	Performance results on IBM SP and SGI Origin.	60
8	Performance results for adding 4 more processors to Doppler filter processing task to case 2 in Table 6. Time in seconds.	63
9	Performance results for adding 16 more processors to pulse compression and CFAR processing tasks to the case in Table 8. Time in seconds. .	64
10	Performance results of non-threaded implementation for 3 cases of nodes assignments.	75
11	Performance results of threaded implementation for 3 cases of nodes assignments.	76
12	Performance results of non-threaded implementation for adding 4 more compute nodes to the Doppler processing task and 4 more compute nodes to pulse compression task to the case 2 in Table 10.	81

13	Performance results of threaded implementation for adding 4 more compute nodes to the Doppler processing task and 4 more compute nodes to pulse compression task to the case 2 in Table 11.	82
14	Performance results on the Paragon with the I/O embedded in the Doppler filter processing task.	93
15	Performance results on the Paragon with the I/O embedded in the Doppler filter processing task.	94
16	Performance results on the SP with the I/O embedded in the Doppler filter processing task.	95
17	Performance results on the Paragon with the I/O implemented as a separate task.	98
18	Performance results on the Paragon with the I/O implemented as a separate task.	99
19	Performance results on the SP with the I/O implemented as a separate task.	100
20	Performance results on the Paragon with pulse compression and CFAR tasks combined.	103
21	Performance results on the Paragon with pulse compression and CFAR tasks combined.	104
22	Performance results on the SP with pulse compression and CFAR tasks combined.	105
23	Percentage of latency improvement when the Pulse compression and CFAR tasks are combined into a single task.	109
24	Performance results on the Paragon with the I/O implemented as a separate task in which the corner turn and type conversion are embedded in the receive phase. PFS stripe factor = 16.	117
25	Performance results on the Paragon with the I/O implemented as a separate task in which the corner turn and type conversion are embedded in the receive phase. PFS stripe factor = 64.	118

26	Performance results on the Paragon with the I/O implemented in the Doppler filter processing task in which the corner turn and type conversion are embedded in the receive phase.	120
27	Performance results on the Paragon with the I/O implemented in the Doppler filter processing task in which corner turn and type conversion are embedded in the receive phase.	121

List of Figures

1	The effect of parallelism and pipelining on the system throughput. . .	4
2	Basic structure for the computational model of the parallel pipeline system. Task_i is parallelized on P_i processors.	5
3	Three architectures of HPC systems: (a) Symmetrical Multi-Processor, (b) cluster, and (c) Massively Parallel Processor.	13
4	Operation stages performed in two radar signal processing algorithms: (a) pre-Doppler STAP and (b) post-Doppler STAP. A series of CPI data sets represent signals collected in different time intervals.	17
5	Data used for Doppler filter processing and Weight computation in STAP algorithms.	19
6	Implementation of the ruggedized version of Intel Paragon system in RTMCARM experiments.	22
7	Model of the parallel pipeline system. Note that Task_i for all input instances is executed on the same number of processors.	24
8	Execution flow of a single compute node in a parallel pipeline system. For each individual task, there are three phases: receive, compute, and send.	25
9	A pipeline system with spatial data dependency only.	27
10	A pipeline system with both spatial and temporal data dependencies.	28
11	Two types of data redistribution: corner turn and left-right shift. Corner turn involves an all-to-all personalized communication and left-right shift involves a many-to-many communication.	30

12	RTMCARM system block diagram.	35
13	Implementation of parallel pipelined STAP. Arrows connecting task blocks represent data transfer between tasks.	39
14	Partitioning strategy for Doppler filter processing task. The CPI data cube is partitioned among P_0 processors across dimension K	42
15	(a) Staggered CPI data partitioned into easy and hard weight computation tasks. (b) Parallel inter-task communication from Doppler filter processing task to easy and hard weight computation tasks requires different sets of range samples. Data collection needs to be performed before the communication. This can be viewed as irregular data redistribution.	43
16	Partitioning strategy for easy and hard weight computation tasks. Data cube is partitioned across dimension N	45
17	Data redistribution from Doppler filter processing task to easy beamforming task. CPI data subcube of size $\frac{K}{P_0} \times J \times \frac{N_{easy}}{P_3}$ is reorganized to subcube of size $\frac{N_{easy}}{P_3} \times \frac{K}{P_0} \times J$ before sending from one processor in Doppler filter processing task to another in easy beamforming task.	47
18	Partitioning strategy for pulse compression task. Data cube is partitioned across dimension N into P_5 processors.	48
19	Organization of data cubes for all tasks in the STAP algorithm.	50
20	Implementation of timing computation and communication for each task. A double buffering strategy is used to overlap the communication with the computation. Receive time = $t_1 - t_0$, compute time = $t_2 - t_1$, and send time = $t_3 - t_2$	52
21	Performance and speedup of computation time as a function of number of processors for all tasks.	55

22	Throughput and latency for the 3 cases in Table 6. Measured results are obtained from the experiments while estimated results are obtained from applying individual tasks' timing to equations (4) and (5). The unit of throughput is number of CPIs per second. The unit of latency is second.	61
23	Throughput and latency results correspond to the cases in Table 7.	62
24	The architecture of a Symmetrical Multi-Processor system.	66
25	The architecture of an Massively Parallel Processing system with SMP nodes.	67
26	The architecture of an Intel Paragon MP system.	68
27	Implementation of two threads in the compute phase. The main thread signals the second thread to perform its computation. After completion of its computation, the second thread signals back to the main thread.	70
28	Performance of different tasks during the compute phase as a function of the number of compute nodes: (a) execution time, (b) speedups, and (c) threading speedups.	73
29	Estimated and measured values of throughput (number of CPIs per second) and latency (seconds per CPI) for both threaded and non-threaded implementations.	77
30	Performance results of integrated pipeline system for threaded and non-threaded implementations, corresponding to Tables 10 and 11.	78
31	Throughput and latency results by adding 2 nodes at a time to each task.	80
32	Data flow of a radar and signal processing system using parallel computers.	85
33	Four CPI data files are read from the parallel file system into the pipeline system in a round-robin manner.	89
34	I/O task is embedded in the Doppler filter processing task of the STAP pipeline system.	90

35	A separate I/O task for reading CPI data is added to the STAP pipeline system.	91
36	Performance results for the STAP pipeline system with parallel I/O embedded in the Doppler filter processing task. This figure corresponds to Tables 14, 15, and 16.	96
37	Performance results for the implementation using a separate I/O task. This figure corresponds to Tables 17, 18, and 19.	101
38	Performance results for the STAP pipeline system that combines the pulse compression and CFAR tasks into a single task. This figure corresponds to Tables 20, 21, and 22.	106
39	Performance comparison of the pipeline system with and without task combining. The throughput results remain approximately the same. Latency is improved when the last two tasks are combined.	107
40	Raw CPI data received from a phased array radar is used to form a $128 \times 512 \times 16$ three dimensional data cube.	112
41	(a) Raw CPI data received from the radar as a $128 \times 512 \times 16$ data cube. (b) Corned-turned CPI data cube of size $512 \times 16 \times 128$. (c) Raw CPI partitioned among 4 reading nodes. (d) Corned-turned CPI partitioned among 5 nodes.	114
42	Implementation of parallel reading of raw CPI data from disks and its distribution for the Doppler filter processing task.	116
43	Performance results for the implementation using a separate I/O task in which the corner turn and type conversion are embedded in the receive phase. This figure corresponds to Tables 24 and 25.	119
44	Performance results for the implementation when the parallel I/O, corner turn, and type conversion are embedded in the receive phase of the Doppler filter processing task. This figure corresponds to Tables 26 and 27.	122

Acknowledgments

I would like to thank my advisors, Professor Pramod Varshney, Alok Choudhary, and Donald Weiner. I am grateful to Professor Varshney for his help and encouragement throughout the course of this work. Especially during my most difficult time at Syracuse University, he gave me the opportunity of this research project and guided me toward this degree. I would like to thank Professor Choudhary for his expert guidance and intellectual inspiration to shape my research over space and time. His valuable comments and suggestions have made this work all more complete. My thanks also go to Prof. Weiner whose wisdom and ideas in the area of radar signal processing formed the foundations of this work.

I am grateful to Russell Brown, Mark Linderman, Richard Linderman, and Zen Pryk for their help, support, and encouragement during the course of this work. Thanks are also due to Dr. Nagaraj Shenoy for his valuable advice.

I sincerely thank my parents for their unwavering love and support throughout these years. Much gratitude goes to my my wife, Pei-hsun, for her patience and encouragement at every step of this long journey, and our beloved son, Justin, for bring all the joy and luck since the day he was born. I thank my sister who have been always prayed for me these years.

I gratefully acknowledge use of the Intel Paragon at California Institute of Technology and the IBM SP at Argonne National Laboratory. This work was supported by Air Force Materiels Command under contract F30602-97-C-0026.

Chapter 1

Introduction

A key requirement for Moving Target Indicator (MTI) radars is that they are small in size, light in weight, and consume low power so that they can be installed on platforms, such as airborne radars, undersea sonars, and ground moving radar stations. In order to fulfill these requirements, radar signal processing systems were traditionally built by using custom VLSI circuits. Most of these Application Specific Integrated Circuits (ASICs) are contemporary non-commercial products designed for special purposed uses. However, the use of non-commercial products results in a higher cost of system development. To reduce the cost, system designers nowadays tend to use Commercial-Of-The-Shelf (COTS) products because they offer lower cost hardware, faster development, and higher reliability while adhering to the size, weight, and power requirements [1].

High Performance Computing (HPC) systems are becoming a feasible alternative due to the progress made in hardware as well as software support in the last few years. Since HPC systems are made of COTS products, they are replacing the custom VLSI based radar systems. Furthermore, HPC systems offer advantages of affordability, scalability, reusability, and flexibility over the traditional custom VLSI based solutions. These powerful machines have traditionally been used to solve scientific problems which require a large number of computational operations. There are many radar signal processing applications such as Space-Time Adaptive processing

(STAP) that are computational intensive and must operate in real time [2]. Such applications may benefit from the use of HPC systems. This dissertation investigates the use of HPC systems for the STAP application. Specifically, a new computational model is developed and performance enhancement achieved by this model is demonstrated.

1.1 Parallelism in STAP

To design a radar signal processing system on HPC platforms, we first have to understand the parallelism embedded in the radar applications. In this work, we will focus on the investigation of parallelization and performance evaluation for STAP algorithms. These algorithms have of considerable interest to the radar signal processing community for some time. They present a challenge to signal processing systems which are required to operate in real time.

STAP applications entail filtering, convolution and correlations, inner and outer products, solvers (direct or iterative), Fast Fourier Transforms (FFTs), etc. A majority of individual processing steps in this application domain exhibit Single Program Multiple Data (SPMD) parallelism. Therefore, an overall integrated system can be thought of as a collection of communicating SPMD programs, that is, at a higher level there exists task parallelism. This parallelism can be further divided into two categories; namely, spatial and temporal parallelism. Spatial parallelism refers to parallel computation on data from the same time instance. Temporal parallelism refers to the performance of tasks on data from different time instances [3, 4, 5].

Ideally, the parallelization of a STAP application must be done by partitioning all the computational load evenly across the processors to achieve maximum efficiency. However, this strategy may not be feasible for the type of STAP applications to be parallelized in a real HPC environment. From the hardware point of view, the architecture of a computer system basically consists of three most essential components: CPU, cache memory, and main memory. Compared to the main memory, the cache is a small capacity and high speed memory used as a buffer between a CPU and the

main memory. The purpose of a cache memory is to reduce the time the CPU must spend waiting for data to arrive from the slower main memory. When a memory request is generated by the CPU, the request is first presented to the cache, and if the cache cannot respond, the request is then presented to the main memory. The cache memory is usually divided into instruction cache and data cache. Since a STAP application has many processing steps that are executed in sequence on the input signal data, this application contains several different instructions sets. If a STAP application is parallelized on all processors in a HPC system, each processor must execute all the instruction sets which results in a poor utilization of the instruction cache. The effect of cache miss can become a significant execution overhead in the real computational environment.

An alternative implementation strategy for a STAP application is called pipelining. The idea of pipelining is that the rate of execution of instructions can be increased by overlapping the execution of different instructions among processors. The pipelining technique can achieve the same throughput results as parallelization on all processors. Consider a job with workload W executed on P processors. The parallelization on all P processors produces a throughput of $\frac{P}{W}$. The pipelining technique can achieve the same throughput after the pipeline is filled which takes W time. Figure 1 illustrates the effects of the two implementations on the throughput result.

For most of the radar applications including STAP, the input data is an indefinite sequence of signal data sets collected by the sensors to be processed by the signal processing system. In addition, a STAP application has a sequence of processing steps in which the output of each step is the input of its successor. Therefore, the pipelining implementation is more appropriate for the design of a STAP system due to the nature of the STAP application. From the hardware point of view, the utilization of instruction cache for every processor in a pipeline can be enhanced by executing only one set of instructions. However, the communication overhead due to data transfer between processors in the pipeline may cause a performance degradation for the integrated system. In the real STAP application, this overhead can be tolerated since the size of each signal data set is moderate compared to many scientific applications.

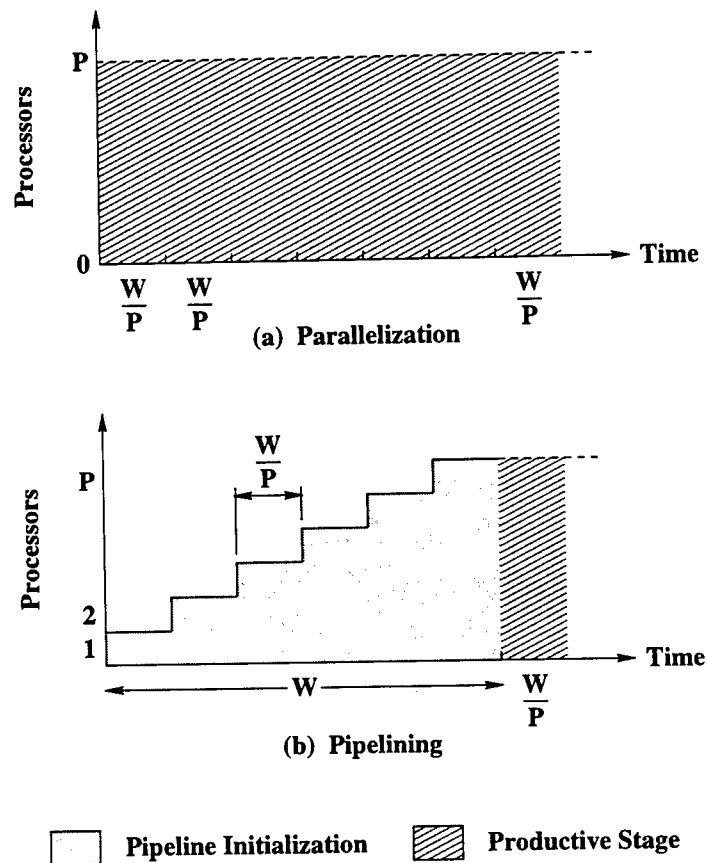


Figure 1. The effect of parallelism and pipelining on the system throughput.

Also, the communication costs can be reduced by overlapping communication and computation in the pipeline.

In this work, we design a parallel pipeline computational model for STAP which combines both the parallelization and pipelining techniques. The pipeline in this model consists of several tasks where each task represents a processing step in a STAP algorithm. The input of one task in the pipeline is the output of its previous task. Each task is then parallelized by partitioning its workload evenly across the processors assigned to this task. Since many of the same operations in a task are repetitively performed on its input data and the size of data for each operation is relatively small, the sequential version of these operations is used instead of using a

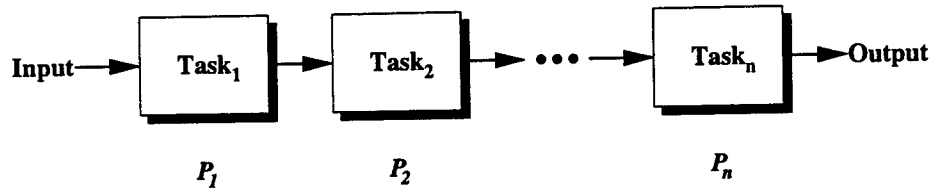


Figure 2. Basic structure for the computational model of the parallel pipeline system.
Task_{*i*} is parallelized on P_i processors.

parallel algorithm. In this manner, the communication between processors within a single task is reduced to a minimum. Figure 2 illustrates the basic structure of the parallel pipeline computational model. In this work, a real STAP application based on this model was implemented on several HPC systems and the performance results demonstrate the linear scalability of this model.

1.2 Shared Memory versus Distributed Memory HPC Systems

Shared-memory multiprocessor architectures are the oldest form of parallel processing architecture. All processors in a shared-memory system have equal access to the system memory through a system bus. The shared-memory architecture is favored because it is the most affordable way to achieve scalability by plugging processors into the system board and then providing an improved performance. There is no inter-processor communication overhead in this system because processors communicate with each other by accessing the common memory. However, there are disadvantages for these types of systems. First, a cache is associated with each processor, which raises the problem of cache coherence. A given piece of data which refers to an address in the main memory can be present at the same time in several caches, so when a processor updates data, a cache-coherency protocol must prevent other processors from accessing the non-updated copies of that piece of data. On the

Table 1. Characteristics of shared-memory and distributed-memory HPC systems.

	Shared-Memory System	Distributed-Memory System
Processors	Multiple	Multiple
Communication	Access via shared memory	Message passing
Operating System	A single copy across all processors	One copy per node
Interconnection	System bus	Data network
Overhead	Access to common data structures	Communication costs

hardware side, the parallelism is restricted by the number of processors that can be connected to a single system bus and the cache/memory transfers and cache-coherency traffic increase with the number of processors. On the software side, the operating system must be designed such that multiple accesses to common data structures are protected. The execution streams on processors must synchronize their accesses to common data by using locks to prevent simultaneous updating.

Distributed-memory massively parallel processor architecture is defined as a potentially large set of compute nodes linked by a specialized inter-processor connection network. The interconnection network provides a scalable bandwidth with a very low latency; e.g., in the order of a few tens of microseconds. Each compute node has its own processor and private memory as well as its own copy of the operating system. Compute nodes in this type of a system communicate with each other through message passing over the interconnection network. Compared to the shared-memory system, the inter-processor communication overhead may become large enough to degrade the performance of the distributed-memory system. Table 1 summarizes the characteristics of these two multiple processor architectures.

A hybrid system that combines both the characteristics of shared and distributed-memory architectures is implemented on Intel Paragon MP system. In the Paragon

MP system, a large number of compute nodes are interconnected by a high speed fabric data network and each compute node is a shared-memory system with a small number of processors. The communication between compute nodes is done by message passing through the data network. In this work, we investigate the implementation of the parallel pipeline STAP system on the Paragon MP system. By using the multi-threading technique, we evenly divide the computational workload across all the processors in every single compute node. Our goal is to determine the performance enhancement that can be obtained using a small number of shared-memory processors at each compute node.

1.3 Disk I/O

Since most radar applications require signal processing in real time, we assume that the signal data collected by the radar is directly delivered to the signal processing system. In addition, the output of the processing system is assumed to be sent to a terminal that displays the detected target in real time. Therefore, the parallel pipeline STAP system normally does not include disk I/O costs. However, with the recent advances in both hardware and software aspects of parallel I/O techniques, parallel file systems can provide powerful disk I/O performance. High data transfer bandwidth between disks and computing systems is achieved by using file systems with multiple stripe directories. In this work, we also investigate the impact of I/O on the overall parallel pipeline STAP system performance when the signal data is obtained through parallel file systems.

1.4 Processor Assignment

Optimal use of resources is particularly important in high-performance embedded applications due to limited resources and other constraints such as desired latency or throughput [6]. The throughput requirement says that when allocating processors to tasks, it should be guaranteed that all the input data sets will be handled in a timely

manner. That is, the processing rate should not fall behind the input data rate. The response time criteria, on the other hand, require the minimization of computation latency on a particular set of data input.

In this work, we investigate the effect of processor assignment in the parallel pipeline system on the throughput and latency. Throughput can always be improved by increasing the number of processors assigned to the task with the maximum execution time among all the tasks in the pipeline system. From the study of the experimental results, the latency may be improved by reorganizing the task structure in the parallel pipeline system. With the fixed total number of processors, tradeoffs exist between assignment of processors to maximize the overall throughput and assignment of processors to minimize the latency. We will examine the possibility of improving the results for both latency and throughput.

1.5 Organization of the Dissertation

This dissertation is organized as follows. Chapter 2 first discusses the traditional design of radar signal processing systems and states the advantages of HPC based systems over VLSI based systems. Then, we propose the parallel pipelined computation model for real time signal processing applications, especially for STAP applications. By analyzing the computational characteristics of STAP algorithms, we describe the structure of this model with respect to the characteristics found in the STAP algorithm and also discuss some parallelization issues and approaches.

In Chapter 3, the implementation of a modified PRI-staggered post-Doppler STAP application based on our parallel pipeline computational model on HPC systems is described. A general description of the STAP algorithm used is given and its associated parallelization approaches are also presented. We ran the codes portably and obtained the performance results on Intel Paragon, IBM SP, and SGI Origin. We present the performance results with regard to computation and communication costs as well as the overall system throughput and latency.

Chapter 4 describes the multi-threaded implementation of the parallel pipeline

STAP system on the Intel Paragon MP system. We compare the performance results between the systems that use single processor nodes and the ones that use SMP nodes. We also discuss the limited performance gain resulting from using the thread-safe versions of numerical libraries developed on the Paragon MP system. The decision of processor assignment is studied when extra compute nodes are available to be added into the pipeline system. This decision presents the tradeoff between increasing throughput and reducing latency.

In Chapter 5, we study the effect on system performance when disk I/O is incorporated in the parallel pipeline model. Two alternatives for I/O implementation are presented: embedding I/O in the pipeline or having a separate I/O task. We use the parallel file systems on the Intel Paragon and the IBM SP to perform the parallel I/O and study its effect on throughput and latency. From these two I/O implementations, we discover that the latency may be improved when the structure of the pipeline is reorganized by merging multiple tasks into a single task. The possibility of improving latency is also examined.

Chapter 6 gives conclusions and outlines further research topics.

Chapter 2

Parallel Pipeline System Model

Traditional radar signal processing systems have been built using custom VLSI in order to meet real-time requirements. Custom VLSI based systems satisfy the requirements for small size and light weight such that they can be installed on airborne platforms. However, custom VLSI can only be used for special purpose radar equipments and have to be re-designed when different signal processing algorithms are to be employed. Due to the technological advances in recent years, High Performance Computing (HPC) systems that provide the necessary computational power to solve many scientific problems are becoming practical. These systems offer advantages of programming generality, software portability, architectural flexibility, affordability, and performance scalability over the systems using traditional custom VLSI. Equipped with powerful processors, HPC systems are attractive in real-time embedded environments, such as radar signal processing applications.

Most radar applications such as Space-Time Adaptive Processing (STAP) are computationally intensive and must operate in real time which require performance levels of Tera Floating point Operations Per Second (TFLOPS) [2]. Since this work focuses on the parallelization of STAP algorithms for HPC environments, we first analyze the computational requirements of STAP algorithms. Based on the computational characteristics of STAP algorithms, we have designed a parallel pipeline computational model. This model permits us to significantly improve the performance of STAP

applications on HPC systems.

This chapter is organized as follows. Section 2.1 compares the approaches of using custom VLSI and HPC systems in radar signal processing applications. Section 2.2 presented an overview of STAP algorithms and describes the computational characteristics of most STAP applications. Section 2.3 discusses the related work on parallelization of STAP applications in the current literature. The parallel pipeline model is presented in Section 2.4. The parallelization issues of this model are given in Section 2.5.

2.1 Custom VLSI versus HPC Systems

Traditionally, radar signal processing systems were built based on custom VLSI. In these systems, VLSI was used to implement several hardware components for special purpose operations, for example, Fast Fourier Transforms (FFT) processors, linear algebra solvers, matrix multipliers, etc. This VLSI based systems satisfy the embedded system requirements of small size, light weight, and fit into the limited space on airborne platforms. However, in order to achieve the required computational performance, the software developed for these systems use low level languages. This restricted the programming flexibility provided by these systems. In addition, significant design modification was often required when porting across different application environments. Therefore, the VLSI based design can only suffice for one signal processing application on one VLSI board. Due to this application specific nature, the hardware development for each individual application leads to higher overall costs.

In recent years, the radar signal processing area has evolved significantly. Many new algorithms have been proposed and developed to solve detection problems in different environments. Most of these algorithms such as STAP require higher computational power in the order of TFLOPS. Adequate processing power is not provided by the current custom VLSI based systems to perform these computations in real time. Hence, different approaches for the design of embedded real time signal processing systems are needed.

High performance computing (HPC) systems are becoming mainstream due to the progress made in hardware as well as software support in the last few years. These powerful machines were used to solve scientific problems, such as grand challenge problems, which require a large number of computational operations. The current commercial HPC systems are developed by integrating commercial-off-the-shelf components interconnected by a high speed data network. Typically, a single processor in HPC systems offers performance in the range 100 Mega Floating point Operations Per Second (MFLOPS) to 600 MFLOPS. Different architectures of HPC systems are implemented such as the classical Symmetrical Multi-Processor (SMP), cluster, or Massively Parallel Processor (MPP) systems.

SMP systems were introduced for mainframe systems during the 1960s. The SMP architecture is favored because of its affordability to achieve scalability by simply plugging in additional processor boards and thus providing an improved performance. From the hardware point of view, an SMP system may contain more than one processor. Every processor in an SMP system has its own private cache memory but shares all other system resources such as the main memory and I/O. Figure 3(a) illustrates the architecture of an SMP system. As opposed to SMPs, clusters are loosely coupled computers where each member computer of the cluster is a system with fully features that is able to function independently from the others. The interconnection in a cluster system is based on a high speed Local Area Network (LAN), for example, Ethernet or FDDI. Figure 3(b) illustrates the architecture of a cluster HPC system. An MPP configuration can be defined as a potentially large set (several hundreds) of CPUs called nodes, linked by a specialized interconnection network. Such an interconnection is based on proprietary technology of individual manufacturers which provides scalable bandwidth with a very low communication latency. Each node has its own processor(s), memory, and I/O channels, as well as its own copy of the operating system. Figure 3(c) illustrates the architecture of an MPP system.

In contrast with the custom VLSI based systems, HPC systems offer advantages of architectural flexibility, performance scalability, and software portability. HPC system architectures can be scaled in size to suit the requirements of radar applications.

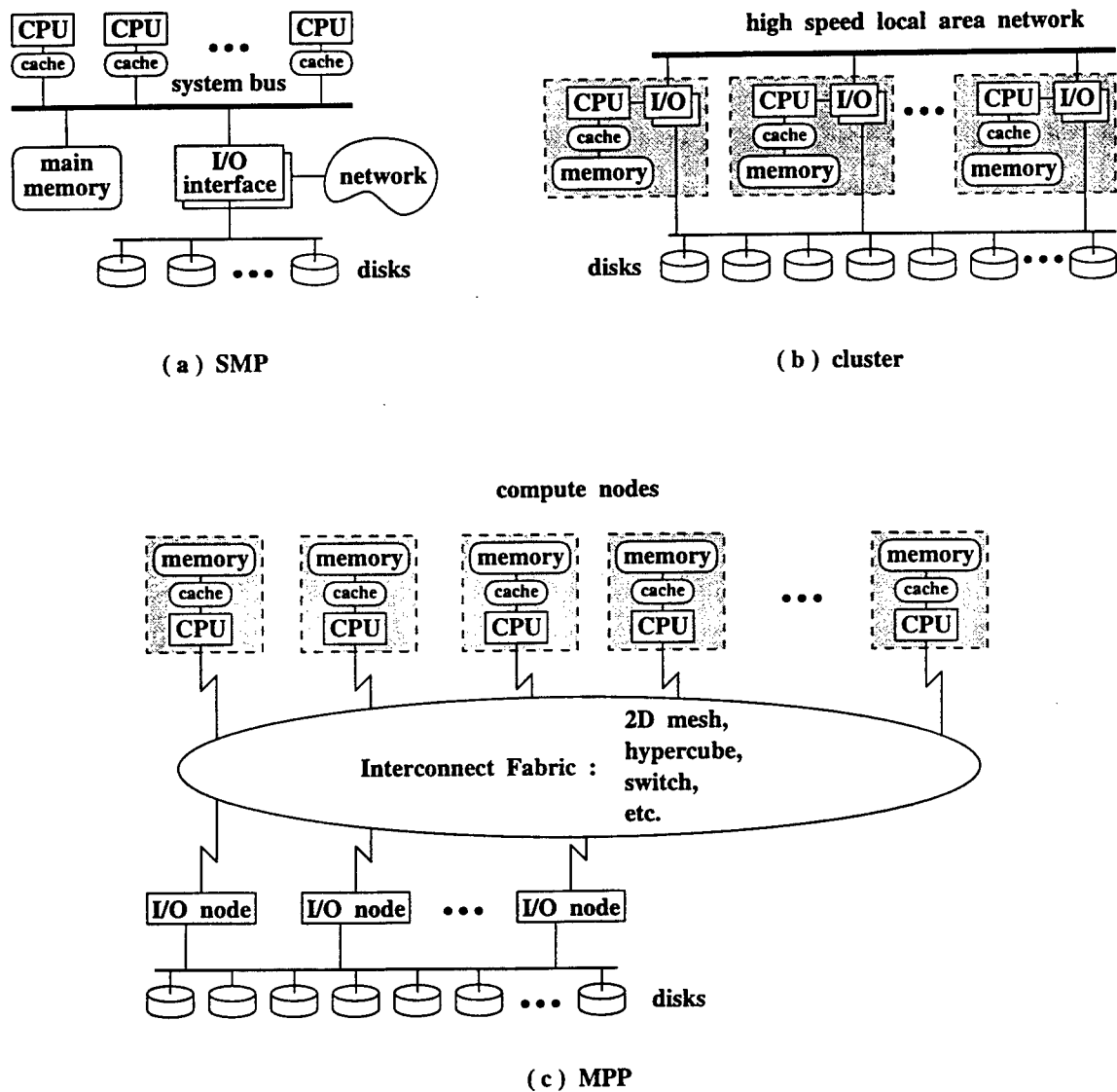


Figure 3. Three architectures of HPC systems: (a) Symmetrical Multi-Processor, (b) cluster, and (c) Massively Parallel Processor.

Table 2. Feature comparison between custom VLSI and HPC systems.

	Custom VLSI	HPC Systems
Hardware components	Special purpose	General purpose
Computation	Fine grain	Coarse grain
Communication	Systolic	Message passing or shared memory
Software design	Low level language	High level language
Architecture flexibility	No	Yes
Scalability	No	Yes
Portability	No	Yes
Affordability	No	Yes

On these HPC systems, software can be implemented in high level programming languages. Use of standardized programming languages and software libraries such as Message Passing Interface (MPI) [7] permits easy portability across various platforms. With recent advances in micro-system technologies, it is now feasible to manufacture light weight, small size, and low power versions of these HPC computers for real-time embedded radar systems. Table 2 compares the features of custom VLSI and current HPC systems.

Digital signal processing is one of the core technologies central to the operation of radar systems. Most signal processing algorithms are characterized by the need for high-performance and involve repetitive, numerical-intensive tasks which are ideally suitable to be parallelized on HPC systems. For embedded real time radar signal processing applications, we have designed a parallel pipeline computational model on the HPC systems. This model was designed for these applications to meet their low latency and high throughput requirements. Our work focused on the parallel implementation and performance evaluation of this model for a Space-Time Adaptive processing (STAP) algorithm. STAP is a typical class of signal processing methods whose parallelization is highly desirable. In the next section, we briefly describe the

STAP algorithms and study the computational characteristics of these algorithms.

2.2 Overview of Space-Time Adaptive Processing

The basic purpose of a radar (radio detection and ranging) is to detect the presence of an object of interest and provide information concerning that object [8]. A radar device transmits a waveform into the atmosphere and then listens for the echoes as the transmitted waveform reflects back from surrounding objects. Various types of information, such as range, velocity, angular coordinates, size, etc., can be obtained from the incoming echoes to detect the desired targets. Range information can be inferred from the amount of time it takes the transmitted signal to travel to a target and then arrive back at the receiver. Directional information can be attained by scanning the surrounding space with a directed beam. Velocity or target movement can be determined through measuring the Doppler shift induced in the reflected waveform. Relative motion between a signal source and a receiver creates a Doppler shift of the source frequency. When a radar system intercepts a moving object that has a radial velocity component relative to the radar, the reflected signal's frequency is shifted. Moving-target indication (MTI) radar is a special purpose Doppler radar that is designed to measure the shift between the transmitted frequency and the frequency of reflections received from possible targets. The MTI radar rejects signal returns from stationary or unwanted slow-moving targets, such as buildings, hills, tree, sea, and rain, and retains detection information on moving targets such as aircrafts and missiles [9].

In our work, we focus on the study of Space-Time Adaptive Processing (STAP) algorithms which refer to a class of radar signal processing methods that operate on data collected from a set of sensors over a given time interval [2]. The object of these methods is to extract the desired signal from potential target returns comprised of Doppler shifts resulting from radar platform motion, clutter returns, and interference including jamming and sensor noise. The sections that follow provide a brief overview of the numerical operations and computational characteristics of general

STAP algorithms. For a thorough theoretical analysis of STAP, the reader is referred to [2, 10, 11].

2.2.1 STAP Algorithms

The computational requirements to determine the optimal solution for STAP problems are in the order of 10^9 to 10^{12} FLOPS which is too large to process in the allotted time for real-time operations. With the computational power of current HPC systems, this method is not completely feasible within the time deadline. Hence, various heuristic methods have been developed which attempt to approximate the optimal solution while reducing the computational requirement.

Generally speaking, a heuristic STAP algorithm consists of five primary processing stages: Doppler filter processing, adaptive weight calculation, beamforming, pulse compression, and Constant False Alarm Rate (CFAR) processing. The input data to the STAP algorithms are serial sets of echo returns collected by a radar in a sequential manner. Each data set is composed of range, pulse, and channel digital samples. Consequently, a three-dimensional data cube represents each set of STAP input data which is commonly referred to as a Coherent Processing Interval (CPI.) Each CPI data is processed through all stages of the STAP algorithms but only a portion of CPI data may be actually needed in the heuristic approach. Various STAP algorithms exist and the difference is basically the order of processing stages that the CPI data goes through. An overview of the STAP process for pre-Doppler and post-Doppler architectures is shown in Figure 4.

The pre-Doppler STAP performs a reduced dimension space-time adaptive nulling function through adaptive combination of the input antenna element data. This algorithm is classified as pre-Doppler STAP because the beamforming is performed on the data prior to Doppler filter processing. Figure 4(a) illustrates the processing function chain for this architecture. The post-Doppler STAP performs a reduced dimension space-time adaptive nulling function through adaptive combination of the input beam space data and is classified as post-Doppler STAP because beamforming

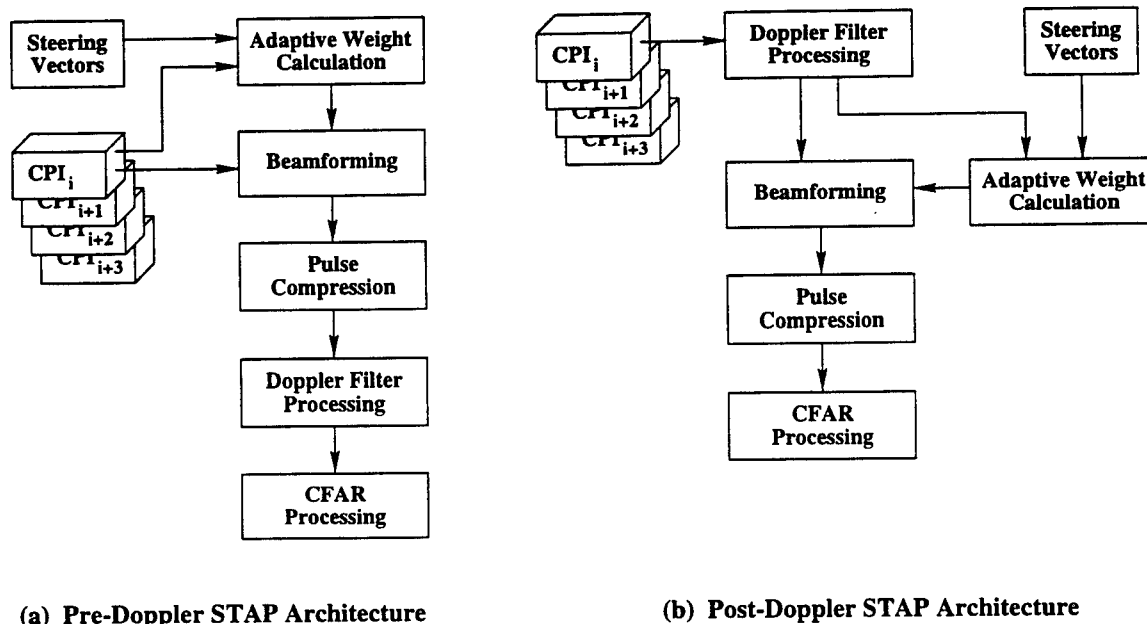


Figure 4. Operation stages performed in two radar signal processing algorithms: (a) pre-Doppler STAP and (b) post-Doppler STAP. A series of CPI data sets represent signals collected in different time intervals.

is performed after the Doppler filter processing. Figure 4(b) illustrates the processing function chain for this architecture. The major operations for all processing stages in STAP are described as follows.

Doppler filter processing The Fast Fourier Transform (FFT) operations are used to transform the CPI data to the Doppler domain. There is an FFT operation for every range and channel. The size for each FFT is equal to the number of pulses in a CPI data cube.

Adaptive weight calculation For each bin in the Doppler domain, the weight vectors are calculated by solving the least squares problem on the matrix of training data. The training data is selected from all range gates to provide a good estimate of the interference. The least squares solution consists of Quadratic

Residue (QR) decomposition and back substitution where the QR decomposition is the most computational intensive operation in a STAP algorithm.

Beamforming The resulting weight vectors are then applied to the CPI data for every Doppler bin. The application is performed by matrix-matrix multiplication.

Pulse compression In this step, the received signal is convolved with a replica of the transmit pulse waveform. This is performed with forward FFTs, vector-vector pointwise multiplications, and inverse FFTs for every Doppler bin and receive beam.

CFAR processing Various CFAR techniques have been developed to detect targets by comparing radar signal returns to an adaptive threshold such that a constant false alarm rate is maintained. Cell-averaging CFAR (CA-CFAR) and ordered statistics CFAR (OS-CFAR) are two well-known methods where CA-CFAR is often used for a homogeneous environment of non-stationary Gaussian noise and OS-CFAR is used for the non-homogeneous environment of Gaussian noise. The operations involved in CA-CFAR are additions and multiplications while OS-CFAR involves sorting which is more computational intensive.

2.2.2 Computational Characteristic Analysis

Some general computational characteristics exist in most of the STAP algorithms [10]. We now examine these characteristics and the potential parallelization strategies.

- A STAP algorithm contains a sequence of processing steps or stages. Each step uses many identical operations on its input data. These identical operations can be performed independently on different sub-sets of data. Since many of these operations are carried out repetitively, an efficient parallelization strategy is to partition the input data across processors such that these operations can be performed sequentially in each processor without communicating with the

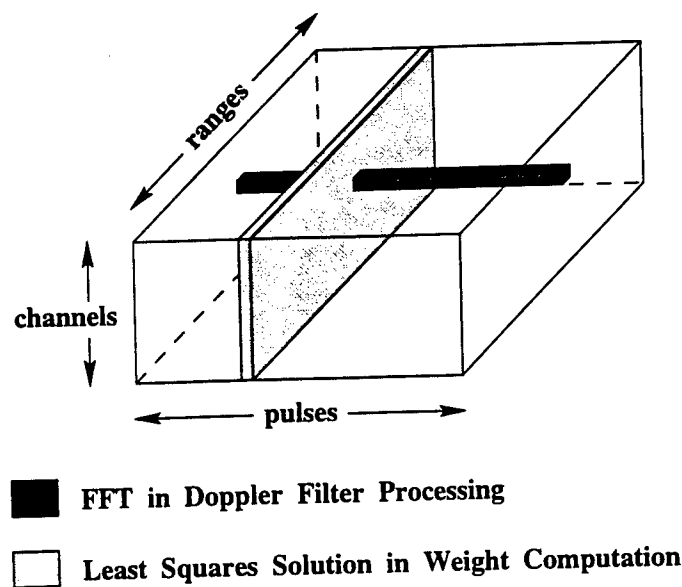


Figure 5. Data used for Doppler filter processing and Weight computation in STAP algorithms.

other nodes. For instance, during Doppler filter processing, several FFTs of the same size are evaluated. Instead of using a parallel algorithm for each FFT, many sequential FFTs are computed in each processor.

- The data access pattern in one step may be different from that of the subsequent step. Each input data set collected by the radar is organized as a three-dimensional complex data cube (CPI data cube.) In the Doppler filter processing, data for the FFTs is distributed orthogonally in the cube relative to the data for solving the least squares solutions in the weight computation. Figure 5 illustrates the data access patterns for these two steps. However, parallelization can only be done by partitioning the data cube across several processors along one of the three axes. Therefore, corner turn on the data cube is required between these two steps. The strategy for data redistribution among processors to achieve the operations of corner turn is critical due to the possibility of having high communication overhead between the two steps.

- The size of each input data set at each step of STAP is moderate and much smaller than those found in grand challenge scientific applications. Normally, the size of each input data set for radar signal processing applications ranges from 1M to 100M bytes. The communication overhead due to parallelization is most likely embedded in the data redistribution between two consecutive steps and its cost depends on the size of the data processed in these two steps. With moderate size of data in most STAP applications, the primary communication will concentrate on the data transfer between two tasks whose access patterns to the same set of data are orthogonal to each other.
- A sequence of signal data sets received by a radar arrives in a continuous fashion and each data set has to be processed through all steps in the STAP algorithm. Data processed in the sequence of steps is like a systolic pipeline that operates on one input data set after another. Parallelization can be done such that one processor performs only one of these steps in order to keep the execution simple and efficient. The radar signal processing applications have a serial or chain-like structure which make themselves amenable to pipelining. In this work, we have designed a model of the parallel pipeline system. This model treats each step as an individual task and assigns groups of processors exclusively to all the tasks to perform the computation in parallel. The details of this model will be discussed in the next section.
- The most important requirements for radar signal processing applications are throughput and latency. The throughput requirement says that all the input data sets should be handled in a timely manner. That is, the processing rate should not fall behind the incoming data rate. The latency criteria, on the other hand, require the minimization of the response time on a particular set of data input. Furthermore, high throughput provides high accuracy for target detection and low latency represents a real time response. The goal for the design of parallel STAP systems is to be able to keep up with the incoming data rate while minimize the data processing time. Therefore, task scheduling

for each STAP process stages on an HPC systems needs to be done carefully to meet these two requirements.

2.3 Related Work

Since the parallelization of STAP applications is highly desirable, several efforts have been devoted to designing efficient parallel STAP algorithms on HPC systems. In May 1996, Air Force Research Laboratory (AFRL) performed a real time STAP demonstration using a Pulse Repetition Interval (PRI)-staggered post-Doppler STAP algorithm onboard an airborne platform [12, 13]. These experiments were performed as part of the Real-Time Multi-Channel Airborne Radar Measurements (RTMCARM) program. The radar in the RTMCARM experiments was a phased array L-Band radar with 32 elements organized into two rows of 16 each. Only the data from the upper 16 elements were processed with STAP. This data was derived from a 1.25 MHz intermediate frequency (IF) signal that was 4:1 oversampled at 5 MHz. The number representation at IF was 14 bits, 2's complement and was converted to 16 bit baseband real and imaginary numbers. Special interface boards were used to digitally demodulate IF signals to baseband. The signal data formed a raw 3-dimensional data cube called coherent processing interval (CPI) data cube comprised of 128 pulses, 512 range gates (32.8 miles), and 16 channels. These special interface boards were also used to corner turn the data cube so that CPI is unit stride along pulses. It speeds the subsequent Doppler processing on the High Performance Computing (HPC) systems. Live CPI data from a phased-array radar were processed by a ruggedized version of the Paragon computer.

The ruggedized version of Intel Paragon system consists of 25 compute nodes running the SUNMOS operating system. Figure 6 depicts the system implementation. Each compute node has three i860 processors accessing the common memory of size 64M bytes as a shared resource. The CPI data sets were sent to the 25 compute nodes in a round robin manner and all three processors worked on each CPI data set as a shared-memory machine. The system processed up to 10 CPIs per seconds

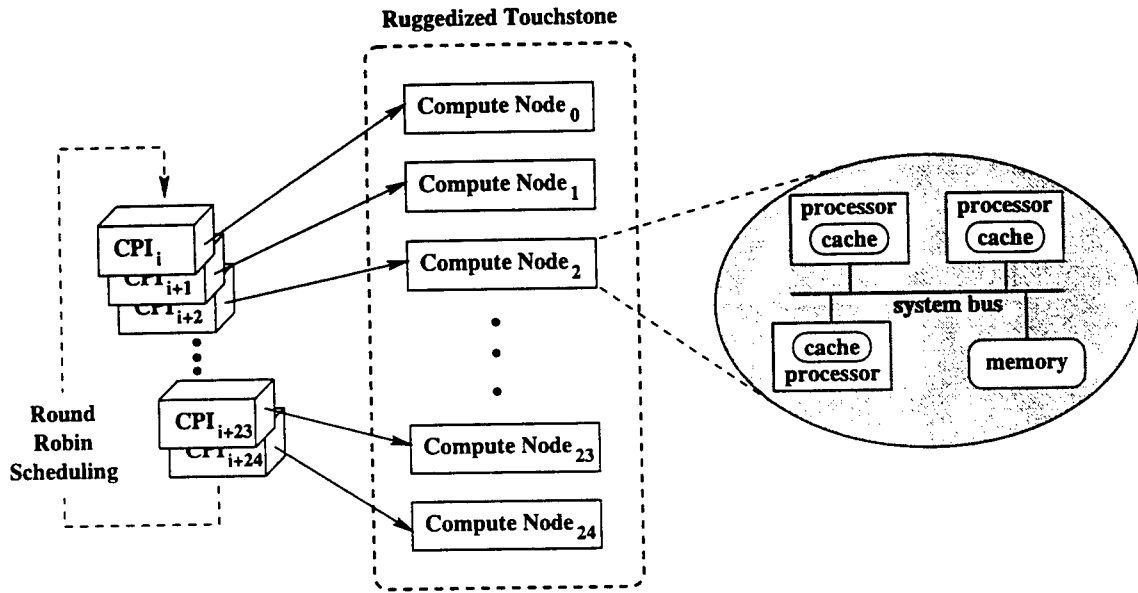


Figure 6. Implementation of the ruggedized version of Intel Paragon system in RTM-CARM experiments.

(throughput) and achieved a latency of 2.35 seconds per CPI. This implementation used compute nodes of the machine as independent resources to run different instances of CPI data sets. No communication among compute nodes was needed. This approach can achieve desired throughput by using as many nodes as needed, but the latency is limited by what can be achieved using the three processors in one compute node. More information on the overall system configuration and performance results can be found in [13, 12].

Other related work such as [10, 14, 15, 16] parallelized high-order post-Doppler STAP algorithms by partitioning the computational workload among all processors allocated for the applications. In [10, 14], they focused on the design of parallel versions of subroutines for FFT and QR decomposition. The total communication time increases as the number of processors increases. The fine grain parallelism generates many short messages for each operation of parallel FFT and QR decomposition and

incurs significant communication overhead. In [15, 16], the implementations optimized the data redistribution between processing steps in the STAP algorithms while using sequential versions of FFT and QR decomposition subroutines. A multi-stage approach was employed in [17] which was an extension of [15, 16]. A beam space post-Doppler STAP was divided into three stages and each stage was parallelized on a group of processors. A technique called replication of pipeline stages was used to replicate the computational intensive stages such that different data instance is run on a different replicated stage. Their effort focused on increasing the throughput while keeping the latency fixed. For other related work, the reader is referred to [18, 11, 19].

2.4 Parallel Pipeline Computational Model

Based on the study of computational characteristics of STAP algorithms, we have designed a computational model of the parallel pipeline system which is suitable for STAP applications. Figure 7 shows this system model and illustrates the computational characteristics found in these applications.

A pipeline is a collection of tasks which are executed sequentially on an input data set. The input to the first task of a pipeline is obtained normally from radar sensors or other input devices with the inputs to the remaining tasks coming from outputs of previous tasks. The set of multiple pipelines shown in the figure indicates that the same pipeline is repeated on subsequent input data sets. Each block in a pipeline represents one parallel task, which itself is parallelized on multiple (different number of) processors. That is, each task is decomposed into subtasks to be performed in parallel. Therefore, each pipeline is a collection of parallel tasks. From the architecture point of view, compute nodes (or nodes) in a HPC system are partitioned exclusively into several groups and every task is assigned a group of compute nodes. The number of nodes assigned to one task may be different from other tasks. In this model, no compute node is assigned to more than one task and at least one node is assigned to a task.

From the single compute node point of view, the execution flow for a task in the

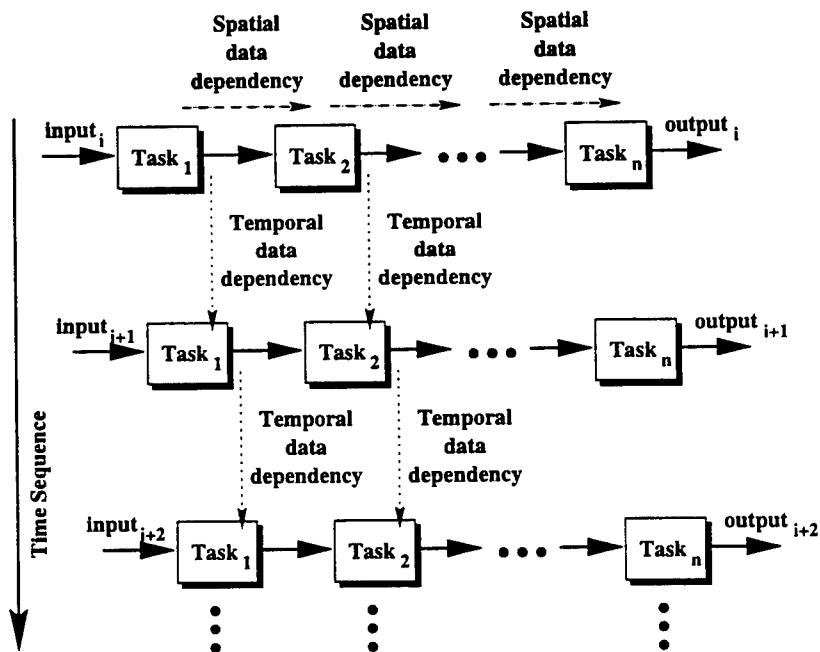


Figure 7. Model of the parallel pipeline system. Note that Task_i for all input instances is executed on the same number of processors.

parallel pipeline consists of three phases: receive, compute, and send phases. During the receive phase in a task, a compute node receives data as the input of this task from the nodes assigned to the previous task. In the send phase, the data resulting from the compute phase is transferred to the nodes in the successor task. Figure 8 illustrates the execution flow of a single compute node in a parallel pipeline system. In both the receive and send phases, communication involves data transfer between two groups of nodes where no common node is in these two groups simultaneously. The communication also involves message packing in the send phase and unpacking in the receive phase. Therefore, data redistribution strategy plays an important role in determining the communication performance. In the compute phase, computational work load in each single task is evenly partitioned among all compute nodes assigned in order to achieve the maximum efficiency. For the parallel systems with

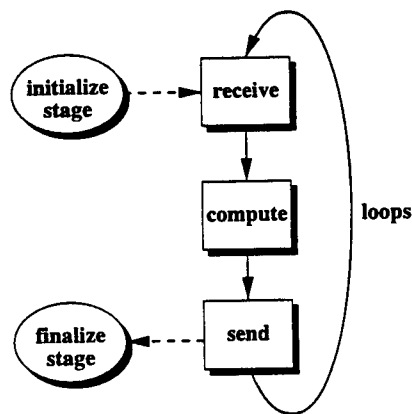


Figure 8. Execution flow of a single compute node in a parallel pipeline system. For each individual task, there are three phases: receive, compute, and send.

multiple processors in each compute node (SMP system), multi-threading technique can be employed to further improve the computation performance. We will discuss the implementation of multiple threads in the parallel pipeline system in Chapter 4.

2.4.1 Data Dependencies

In such a parallel pipeline system, there exist both spatial and temporal parallelism [3, 4, 5]. Spatial parallelism is one in which similar operations are applied in all parts of the input data. That is, the input data for one task can be divided into many granules and distributed to its subtasks which may execute on different processors in parallel. Each task operates on the output data of the previous task as its input and produces an output which becomes the input for the next task. The type of data and data structures may be different for each task in the system but each form of data can be partitioned into several granules to be processed in parallel. Temporal parallelism is present when tasks are repeated on a time sequence of input data sets. The processing of each set of input data can be done in parallel with the processing of data sets of other time instances.

Both spatial and temporal parallelism result in two types of data dependencies

and flows, namely, spatial data dependency and temporal data dependency. Spatial data dependency can be classified into inter-task data dependency and intra-task data dependency. Intra-task data dependencies arise when a set of subtasks needs to exchange intermediate results during the execution of a parallel task in a pipeline. The exchange of data may be needed during the execution of the algorithm, or to combine the partial results, or both. Inter-task data dependency is due to the transfer and reorganization of data passed onto the next parallel task in the pipeline. Inter-task communication can be communication from the subtasks of the current task to the subtasks of the next task, or collection and reorganization of output data of the current task and then redistribution of the data to the next task. The choice depends on the underlying architecture, mapping of algorithms and input-output relationship between consecutive tasks. Temporal data dependency occurs when some form of output generated by the tasks executed on the previous data set are needed by tasks executing the current data set. In the next chapter, we will see that STAP algorithms have both types of data dependencies.

2.5 Parallelization Issues and Approaches

Applications such as STAP entail multiple algorithms (or processing steps), each of which performs particular functions, to be executed in a pipelined fashion. Multiple pipelines need to be executed in a staggered manner to satisfy the throughput requirements. Each task needs to be parallelized for the required performance, which, in turn, requires addressing the issue of data distribution on the subset of processors on which a task is parallelized to obtain good efficiency and incur minimal communication overhead. Given that each task is parallelized, data flow among multiple processors of two or more tasks is required and, therefore, communication scheduling techniques become critical. The problem of input-output of data is another crucial problem and is more challenging in this scenario because data must be redistributed within the pipeline in a timely manner to guarantee the throughput and latency requirements.

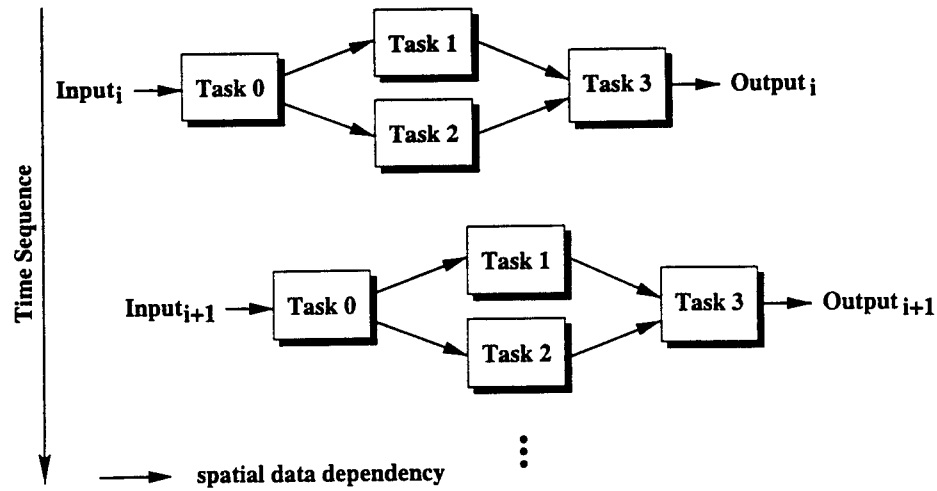


Figure 9. A pipeline system with spatial data dependency only.

2.5.1 Throughput and Latency

Throughput and latency are two important measures for performance evaluation on a pipeline system. Given a parallel pipeline system with n tasks, the throughput of the pipeline system is the inverse of the maximum execution time among all tasks, i.e.,

$$throughput = \frac{1}{\max_{0 \leq i < n} T_i} \quad (1)$$

where T_i represents the execution time of task i . Figure 9 gives an example of a parallel pipeline system with 4 tasks. In this example, the throughput of the pipeline system is

$$throughput_4 = \frac{1}{\max_{0 \leq i < 4} T_i}. \quad (2)$$

To maximize the throughput, the maximum value of T_i should be minimized. In other words, no task should have an extremely large execution time. With limited number of processors, the processor assignment to different tasks must be made in such a way that the execution time of the task with highest computation time is reduced.

The latency of this pipeline system is the time between the arrival of one data set at the system input and the time at which the result data set is available at the

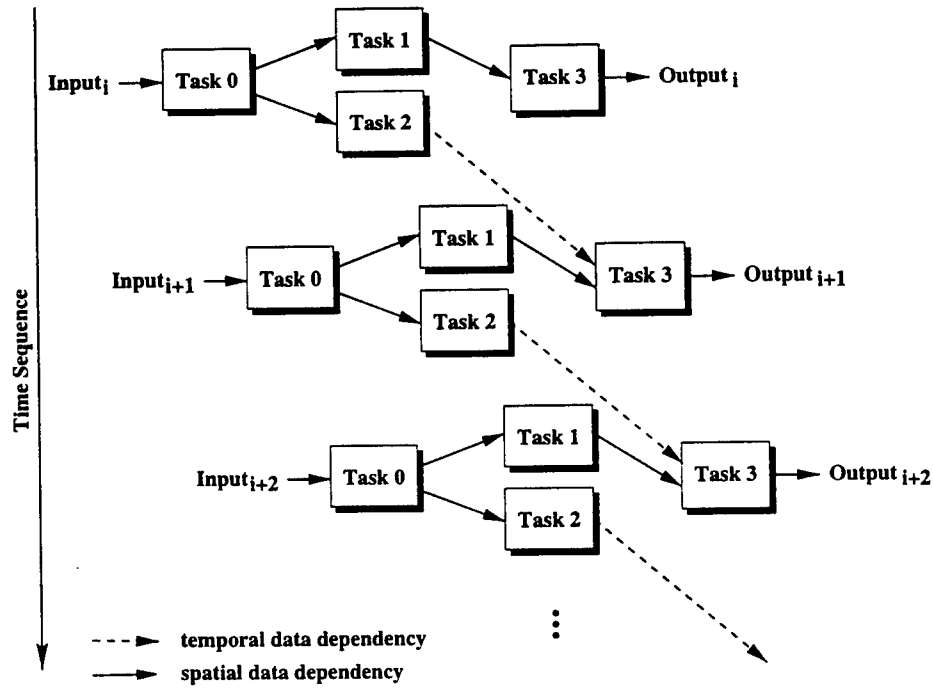


Figure 10. A pipeline system with both spatial and temporal data dependencies.

system output. Therefore, the latency for processing one data set is the sum of the execution times of all the tasks except for the tasks with temporal data dependency, i.e.,

$$latency = \sum_{i=0}^{n-1} T_i - \sum_{i \in TD} T_i \quad (3)$$

Notice that temporal data dependency (TD) exists when a task processes data sets in previous time instances and its results is applying to the data set in current time instance. The temporal data dependency does not affect the latency because the task with temporal data dependency is operating on previous data sets at any time instance and its following tasks do not wait for its completion on current time instant data set. In the example shown in Figure 9, no temporal data dependency exists in the pipeline system and its latency is

$$latency_4 = T_0 + \max(T_1, T_2) + T_3.$$

Figure 10 shows an example of a parallel pipeline system with both spatial and temporal data dependencies. In this example, a temporal data dependency exists between task 2 and task 3. The latency for this pipeline system becomes

$$latency'_4 = T_0 + T_1 + T_3$$

while the throughput is still the same as Equation (1),

$$throughput'_4 = \frac{1}{\max_{0 \leq i < 4} T_i}.$$

To reduce the overall system latency, every parallel task must be allocated more processors to decrease its execution time and consequently the overall execution time of the integrated system.

2.5.2 Data Redistribution

In an integrated system which implements several tasks that feed data to each other, data redistribution is required when it is fed from one parallel task to another. This is because the way data is distributed in one task may not be the most appropriate distribution for another task for algorithmic or efficiency reasons. For example, given an input two-dimensional array, one task may process it in a row major fashion. The next task that receives this two-dimensional array may require a column major order. To ensure efficiency of continuity of memory access, data reorganization and redistribution are required in both intra-task and inter-task communication phase. In inter-task communication, data redistribution also allows concentration of communication at the beginning and the end of each task.

In the parallel pipeline system shown in Figure 7, compute nodes are partitioned into several disjoint groups and each group is assigned to exactly one task in the pipeline. As one group of nodes completes its computation on one set of input data, its output is to be transferred to its successor group or groups, depending on the structure of the pipeline. Data transfer between two tasks represents interprocessor communication between two groups of compute nodes. Since the data access patterns

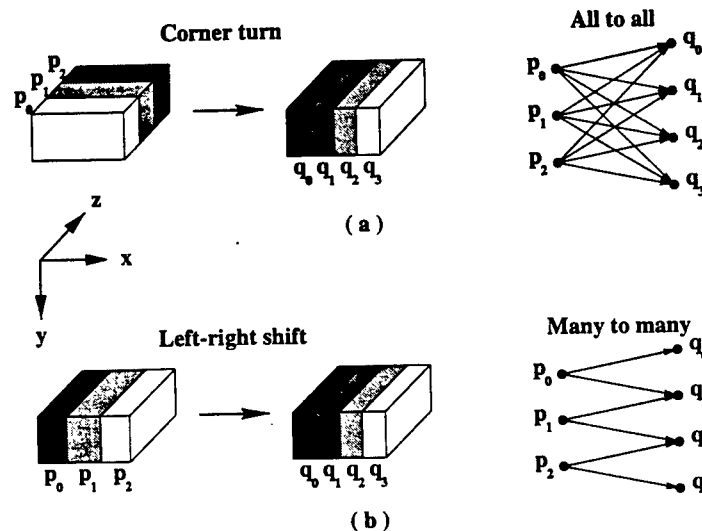


Figure 11. Two types of data redistribution: corner turn and left-right shift. Corner turn involves an all-to-all personalized communication and left-right shift involves a many-to-many communication.

of one task may be different from its successor tasks, communication patterns can be either a corner-turn or a left-right shift pattern.

We now use an example to explain these two types of data redistribution. Given a three-dimensional array as the input to one parallel task, the array is evenly partitioned into several sub-arrays across multiple compute nodes assigned to this task. This partitioning can be done along one of the array's three axes due to the efficiency of memory access. However, after the completion of computation, partitioning of the output array of one group along the same axis may not be suitable for its successor group. Since any single data layout will not always provide efficient computation for data access along two orthogonal axes, the data re-mapping problem exists when intermediate data is transmitted between different parallel tasks. This communication pattern is called a corner-turn communication pattern, shown in Figure 11(a). A three-dimensional array is partitioned along axis z for task p with 3 compute nodes while it is partitioned along axis x for task q with 4 nodes. Therefore, it may be

necessary to reorganize the data during message packing in the send phase of one task and message unpacking in the receive phase of its successor task. The communication pattern of corner-turn data redistribution involves a complete exchange (all-to-all personalized) pattern between two groups of compute nodes.

On the other hand, the left-right shift communication pattern occurs when an array is partitioned along the same axis between two consecutive parallel tasks. In Figure 11(b), a three-dimensional array is partitioned along axis x for both p and q tasks. Left-right shift communication pattern does not involve data reorganization. Each node in one task only communicates with some of the nodes in its successor task (a many-to-many communication.) Therefore, the communication overhead of left-right shift pattern is much less than the corner-turn pattern.

Efficient runtime functions and strategies have been developed to perform data redistribution within the same group of processors [20, 21, 22, 23]. These techniques reduce the communication time on irregular all-to-all redistribution by minimizing node contention. However, our parallel pipeline system model requires data redistribution between two different groups of processors and those algorithms were not designed for the case of our model. In the next chapter, we present several efficient redistribution algorithms which focus on communication between different sets of processors.

2.5.3 Task Scheduling and Processor Assignment

An important factor in the performance of a parallel system, is how the computational load is mapped onto the processors in the system. Ideally, to achieve maximum parallelism, the load must be evenly distributed across the processors. In recent years, much research has been devoted to the problem of mapping large computations onto a system of parallel processors. Various aspects of the general problem have been studied, including different parallel architectures, task structures, communication issues, and load balancing [3, 24]. The problem of statically mapping the workload of a parallel algorithm to processors in a distributed memory system, has been studied

under different problem models, such as [25, 26]. The mapping policies are adequate when an application consists of a single task, and the computational load can be determined statically. These static mapping policies do not model applications consisting of a sequence of tasks (algorithms), where the output of one task becomes the input to the next task in the sequence.

Optimal use of resources is particularly important in high-performance embedded applications due to limited resources and other constraints such as desired latency or throughput [6]. When several parallel tasks need to be executed in a pipelined fashion, tradeoffs exist between assigning processors to maximize the overall throughput and assigning processors to minimize a single data set's response time (or latency.) The throughput requirement says that when allocating processors to tasks, it should be guaranteed that all the input data sets will be handled in a timely manner. That is, the processing rate should not fall behind the input data rate. The response time criteria, on the other hand, require minimizing the latency of computation on a particular set of data input.

To reduce the latency, each parallel task must be allocated more processors to reduce its execution time, and consequently, the overall execution time of the integrated system. But it is well known that the efficiency of parallel programs usually decreases as the number of processors is increased. Therefore, the gains in this approach may be incremental. On the other hand, throughput can be increased by increasing the latency of individual tasks by assigning them fewer processors, and therefore, increasing efficiency, but at the same time having multiple streams active concurrently in a staggered manner to satisfy the input-data rate requirements. We will present these tradeoffs and discuss various implementation issues in the later chapters.

2.6 Summary

This chapter first compared two competing approaches to build a radar signal processing system namely by using custom VLSI and HPC systems. Due to the technological advances in recent years, HPC systems are becoming a practical alternative

that provide the necessary computational power to solve many scientific problems. These systems offer advantages of programming generality, software portability, architectural flexibility, affordability, and performance scalability over the systems using traditional custom VLSI.

An overview of radar signal processing, especially the STAP algorithm, was provided. The computational characteristics embedded in most of the existing STAP algorithms were analyzed. Based on the studies of these characteristics, we designed a parallel pipeline computational model that is suitable for the type of STAP applications. Parallelization issues of this model on HPC systems were also addressed in this chapter. These issues involve data redistribution between two groups of processors and processor assignment among tasks in the pipeline. In the next chapter, we implement a STAP application using our parallel pipeline model to demonstrate the performance efficiency and scalability that this model can achieve.

Chapter 3

Parallel Pipelined STAP System

Space-time adaptive processing (STAP) is a well known technique in the area of airborne surveillance radars used to detect weak target returns embedded in strong ground clutter, interference, and receiver noise. STAP is a 2-dimensional adaptive filtering algorithm that attenuates unwanted signals by placing nulls in their directions of arrival and Doppler frequencies. Most STAP applications are computationally intensive and must operate in real time. High performance computers are becoming mainstream due to the progress made in hardware as well as software support in the last few years. They can satisfy the STAP computational requirements of real-time applications while increasing the flexibility, affordability, and scalability of radar signal processing systems. However, efficient parallelization of a STAP algorithm which has embedded in it different processing steps is challenging, and requires several optimizations.

This chapter describes our parallel pipelined implementation of a Pulse Repetition Interval (PRI)-staggered post-Doppler STAP algorithm. The design and implementation of the application is portable. Performance results are presented for the Intel Paragon at the Air Force Research Laboratory (AFRL), IBM SP at Argonne National Laboratory (ANL), and SGI Origin at Northwestern University. AFRL successfully installed their implementation of the STAP algorithm onboard an airborne platform and performed four flight experiments in May and June 1996 [12]. These experiments

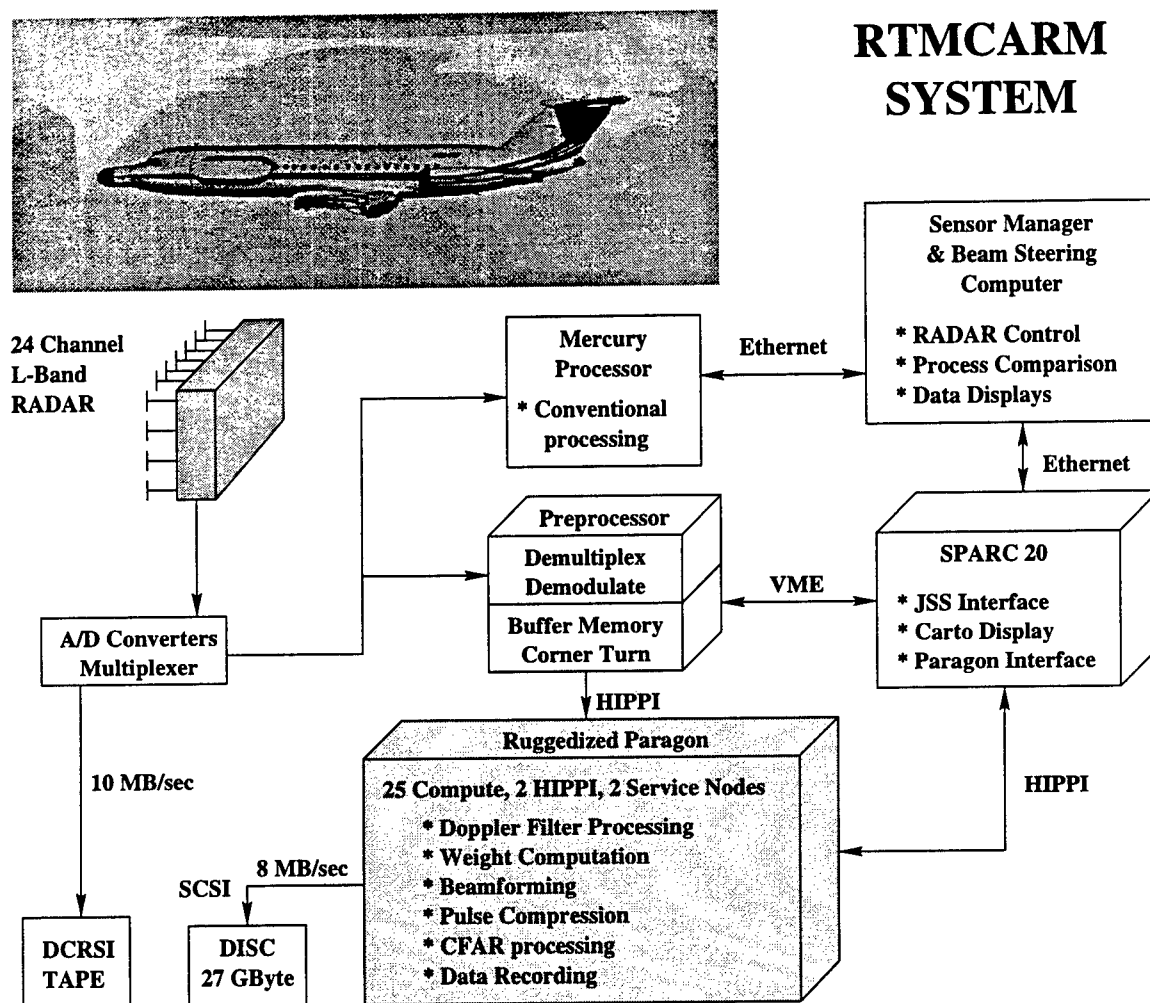


Figure 12. RTMCARM system block diagram.

were performed as part of the Real-Time Multi-Channel Airborne Radar Measurements (RTMCARM) program. The RTMCARM system block diagram is shown in Figure 12. In that real-time demonstration, live data from a phased array radar was processed by the onboard Intel Paragon and results showed that high performance computers can deliver a significant performance gain. However, that implementation used compute nodes of the machine only as independent resources in a round robin fashion to run different instances of STAP (rather than speeding up one instance of

STAP.) Using this approach, the throughput may be improved, but the latency is limited by what can be achieved using one compute node.

For our parallel implementation of this real application we have designed a model of the parallel pipeline system described in Chapter 2 where each pipeline is a collection of tasks and each task itself is parallelized. This parallel pipeline model was applied to the STAP algorithm with each step as a task in a pipeline. This permits us to significantly improve latency as well as throughput.

This chapter describes parallelization process and performance results. In addition, design considerations for portability, task mapping, parallel data redistribution, parallel pipelining as well as system-level and task-level performance measurement are discussed. Finally, the performance and scalability of the implementation for a large number of processors is demonstrated.

This chapter is organized as follows: Section 3.1 presents an overview the STAP algorithm we implemented in our work, Section 3.2 presents specific details of parallel pipeline STAP implementation. Section 3.3 describes the software development and the configurations of system platforms we used in the experiments. Performance results are presented in Section 3.4.

3.1 Algorithm Overview

The adaptive algorithm, which cancels Doppler shifted clutter returns as seen by the airborne radar system, is based on a least squares solution to the weight vector problem. This approach has traditionally yielded high clutter rejection, but suffers from severe distortions in the adapted main beam pattern and resulting loss of gain on the target. Our approach introduces a set of constraint equations into the least squares problem which can be weighted proportionally to preserve main beam shape. The algorithm is structured so that multiple receive beams may be formed without changing the matrix of training data. Thus, the adaptive problem can be solved once for all beams which lie within the transmit illumination region. The airborne radar system was programmed to transmit five beams, each 25 degrees in width, spaced

20 degrees apart. Within each transmit beam, six receive beams were formed by the processor.

The algorithm consists of the following steps:

1. Doppler filter processing,
2. Weight computation,
3. Beamforming,
4. Pulse compression, and
5. CFAR processing.

Doppler filtering is performed on each receive channel using weighted Fast Fourier Transforms (FFT's). The analog portion of the receiver compensates the received clutter frequency to center the clutter frequency at zero regardless of the transmit beam position. This simplifies indexing of Doppler bins for classification as "easy" or "hard" depending on their proximity to mainbeam clutter returns. For the "hard" cases, Doppler processing is performed on two 125-pulse windows of data separated by three pulses (a STAP technique known as "PRI-stagger"). Both sets of Doppler processed data are adaptively weighted in the beamforming process for improved clutter rejection. In the "easy" case, only a single Doppler spectrum is computed. This simpler technique has been termed Post Doppler Adaptive Beamforming and is quite effective at a fraction of the computational cost when the Doppler bin is well separated from mainbeam clutter. In these situations, an angular null placed in the direction of the competing ground clutter provides excellent rejection. Selectable window functions are applied to the data prior to the Doppler FFT's to control sidelobe levels. The selection of a window is a key parameter in that it impacts the leakage of clutter returns across Doppler bins, traded off against the width of the clutter passband.

An efficient method of beamforming using recursive weight updates is made possible by a block update form of the QR decomposition algorithm. This is especially

significant in the "hard" Doppler regions, which are computed using separate weights for six consecutive range intervals. The recursive algorithm requires substantially less training data (sample support) for accurate weight computation, as well as providing improved efficiency. Since the hard regions have one sixth the range extent from which to draw data, this approach dealt with the paucity of data by using past looks at the same azimuth, exponentially forgotten, as independent, identically distributed estimates of the clutter to be cancelled. This assumes a reasonable revisit time for each azimuth beam position. During the flight experiments, the five 25 degree transmit beam positions were revisited at a 1-2 Hz rate (5-10 CPIs per second.)

The training data for the easy Doppler regions was selected using a more traditional approach. Here, the entire range extent was available for sample support, so the entire training set was drawn from three preceding CPIs for application to the next CPI in this azimuth beam position. In this case, a regular (non-recursive) QR decomposition is performed on the training data, followed by block update to add in the beam shape constraints.

Pulse compression is a compute intensive task, especially if applied to each receive channel independently. In general, this approach is required for adaptive algorithms which compute different weight sets as a function of radar range. Our algorithm, however, with its mainbeam constraint, preserves phase across range. In fact, the phase of the solution is independent of the clutter nulling equations, and appears only in the constraint equations. The adapted target phase is preserved across range, even though the clutter and adaptive weights may vary with range. Thus, pulse compression may be performed on the beamformed output of the receive channels providing a substantial savings in computations.

In the sections to follow, we present the process of parallelization and software design considerations including those for portability, task mapping, parallel data redistribution, parallel pipelining and issues involved in measuring performance in implementations when not only the performance of individual tasks is important, but overall performance of the integrated system is critical. We demonstrate the performance and scalability for a large number of processors.

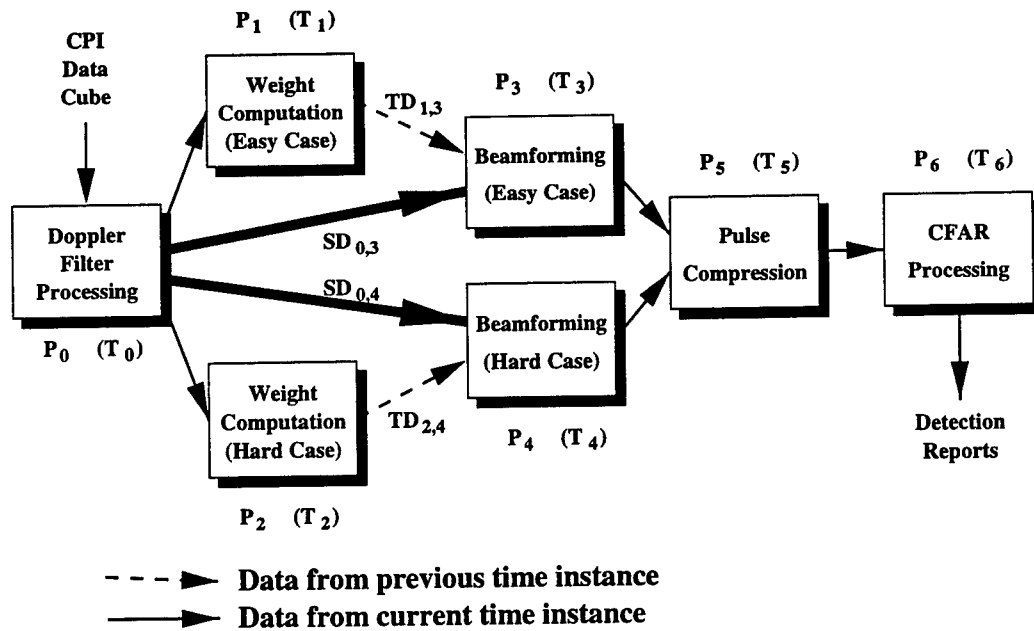


Figure 13. Implementation of parallel pipelined STAP. Arrows connecting task blocks represent data transfer between tasks.

3.2 Design and Implementation

The design of the parallel pipelined STAP algorithm is shown in Figure 13. The parallel pipeline system consists of seven basic tasks. We refer to the parallel pipeline as simply a pipeline in the rest of this chapter. The input data set for the pipeline is obtained from a phased array radar and is formed in terms of a coherent processing interval (CPI). Each CPI data set is a 3-dimensional complex data cube comprised of K range cells, J channels, and N pulses. The output of the pipeline is a report on the detection of possible targets. The arrows shown in Figure 13 indicate data transfer between tasks. Although a single arrow is shown, note that each represents multiple processors in one task communicating with multiple processors in another task. Each task i is parallelized by evenly partitioning its work load among P_i processors. The execution time associated with task i , T_i , consists of the time to receive data from

the previous task, computation time, and time to send results to the next task.

The calculation of weights is the most computationally intensive part of the STAP algorithm. For the computation of the weight vectors for the current CPI data cube, data cubes from previous CPIs are used as input data. This introduces temporal data dependency. For example, suppose that a set of CPI data cubes entering the pipeline sequentially are denoted by CPI_i , $i = 0, 1, \dots$. At any time instance i , the Doppler filtering task is processing CPI_i and beamforming task is processing CPI_{i-1} . In the meanwhile, the weight computation task is using past CPIs in the same azimuthal direction to calculate the weight vectors for CPI_i as described below. The computed weight vectors will be applied to CPI_i in the beamforming task at next time instance $i + 1$. Thus, temporal data dependencies exist and are represented by arrows with dashed lines, $TD_{1,3}$ and $TD_{2,4}$, in Figure 13 where $TD_{i,j}$ represents temporal data dependency of task j on data from task i . In a similar manner, spatial data dependencies $SD_{i,j}$ can be defined and are indicated in Figure 13 by arrows with solid lines.

Throughput and latency are two important measures for performance evaluation on a pipeline system. The throughput of our pipeline system is the inverse of the maximum execution time among all tasks, i.e.,

$$throughput = \frac{1}{\max_{0 \leq i < 7} T_i}. \quad (4)$$

To maximize the throughput, the maximum value of T_i should be minimized. In other words, no task should have an extremely large execution time. With a limited number of processors, the processor assignment to different tasks must be made in such a way that the execution time of the task with highest computation time is reduced.

The latency of this pipeline system is the time between the arrival of the CPI data cube at the system input and the time at which the detection report is available at the system output. Therefore, the latency for processing one CPI is the sum of the execution times of all the tasks except weight computation tasks, i.e.,

$$latency = T_0 + \max(T_3, T_4) + T_5 + T_6 \quad (5)$$

Equation (5) does not contain T_1 and T_2 . The temporal data dependency does not affect the latency because weight computation tasks use data from the previous instance of CPI data rather than the current CPI. The filtered CPI data cube sent to the beamforming tasks do not wait for the completion of its weight computation but rather for the completion of the weight computation of the previous CPI. For example, when the Doppler filter processing task is processing CPI_i , the weight computation tasks use the filtered CPI data, CPI_{i-1} , to calculate the weight vectors for CPI_i . At the same time, the beamforming tasks are working on CPI_{i-1} using the data received from the Doppler filter processing and weight computation tasks. The beamforming tasks do not wait for the completion of the weight computation task when processing CPI_{i-1} data. The overall system latency can be reduced by reducing the execution times of the parallel tasks, e.g., T_0 , T_3 , T_4 , T_5 , and T_6 in our system.

Next, we briefly describe each task and its parallel implementation. A detailed description of the STAP algorithm we used can be found in [27, 28].

3.2.1 Doppler Filter Processing

The input to the Doppler filter processing task is one CPI complex data cube received from a phased array radar. The computation in this task involves performing range correction for each range cell and the application of a windowing function (e.g. Hanning or Hamming) followed by a N -point FFT for every range cell and channel. The output of the Doppler filter processing task is a 3-dimensional complex data cube of size $K \times 2J \times N$ which is referred to as staggered CPI data. In Figure 13, we can see that this output is sent to the weight computation task as well as to the beamforming task.

Both the weight computation and the beamforming tasks are divided into easy and hard parts. These two parts use different portions of staggered CPI data and the associated amounts of computation are also different. The easy weight computation task uses range samples only from the first half of the staggered CPI data while the hard weight computation task uses range samples from the entire staggered CPI data.

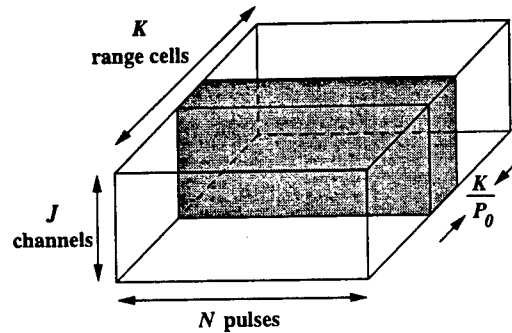


Figure 14. Partitioning strategy for Doppler filter processing task. The CPI data cube is partitioned among P_0 processors across dimension K .

On the other hand, easy and hard beamforming tasks use all range cells rather than some of them. Therefore, the size of data to be transferred to weight computation tasks is different from the size of data to be sent to beamforming tasks. In Figure 13, thicker arrows connected from Doppler filter processing task to beamforming tasks indicates that the amount of data sent to the beamforming tasks is more than the amount of data sent to the weight tasks.

The basic parallelization technique employed in the Doppler filtering processing task is to partition the CPI data cube across the range cells, that is, if P_0 processors are allocated to this task, then each processor is responsible for $\frac{K}{P_0}$ range cells. The reason for partitioning the CPI data cube along dimension K is that it maintains an efficient accessing mechanism for continuous memory space. A total of $K \cdot 2J$ N -point FFTs are performed and the best performance is achieved when every N -point FFT accesses its N data sets from a continuous memory space. Figure 14 illustrates the parallelization of this step. The inter-task communication from the Doppler filter processing task to weight computation tasks is explained in Figure 15(b). Since only subsets of range cells are needed in weight computation tasks, data collection has to be performed on the output data before passing it to the next tasks. Data collection is performed to avoid sending redundant data and hence reduces the communication costs.

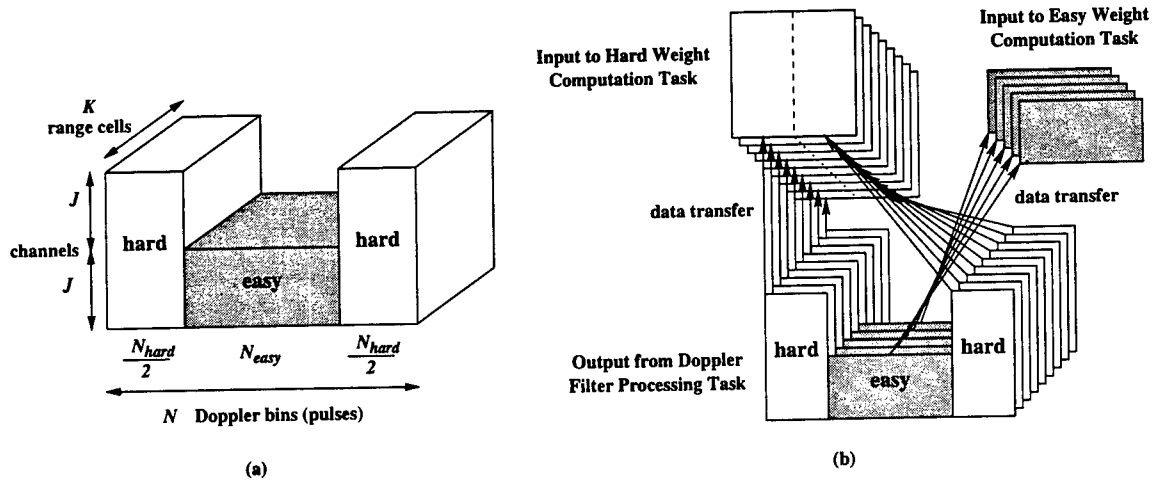


Figure 15. (a) Staggered CPI data partitioned into easy and hard weight computation tasks. (b) Parallel inter-task communication from Doppler filter processing task to easy and hard weight computation tasks requires different sets of range samples. Data collection needs to be performed before the communication. This can be viewed as irregular data redistribution.

3.2.2 Weight Computation

The second step in this pipeline is the computation of weights that will be applied to the next CPI. This computation for N pulses is divided into two parts, namely, "easy" and "hard" Doppler bins, as shown in Figure 15(a). The hard Doppler bins (pulses), N_{hard} , are those in which significant ground clutter is expected. The remaining bins are easy Doppler bins, N_{easy} . The main difference between the two is the amount of data used and the amount of computation required. Not all range cells in the staggered CPI are used in weight calculation and different subsets of range samples are used in easy Doppler bins and hard Doppler bins.

To gather range samples for easy Doppler bins to calculate the weight vectors for the current CPI, data is drawn from three preceding CPIs by evenly spacing out over the first one third of K range cells of each of the three CPIs. The easy weight computation task involves N_{easy} QR factorizations, block updates, and back

substitutions. In the easy weight calculation, only range samples in the first half of the staggered CPI data are used while hard weight computation employs range samples from the entire staggered CPI. Furthermore, range extent for hard Doppler bins is split into six independent segments to further improve clutter cancelation. To calculate weight vectors for the current CPI, range samples used in hard Doppler bins are taken from the immediately preceding staggered CPI combined with older, exponentially forgotten, data from CPIs in the same direction. This is done for each of the six range segments. The hard weight computation task involves $6N_{hard}$ recursive QR updates, block updates, and back substitutions. The easy and hard weight computation tasks process sets of 2-dimensional matrices of different sizes.

Temporal data dependency exists in the weight computation task because both easy and hard Doppler bins use data from previous CPIs to compute the weights for the current CPI. The outputs of this step, the weight vectors, are two 3-dimensional complex data cubes of size $N_{easy} \times J \times M$ and $N_{hard} \times 2J \times M$ for easy and hard weight computation tasks, respectively, where M is the number of receive beams. These two weight vectors are to be applied to the current CPI in the beamforming task. Because of the different sizes of easy and hard weight vectors, the beamforming task is also divided into easy and hard parts to handle different amounts of computation.

Given the uneven nature of weight computations, different sets of processors are allocated to the easy and hard tasks. In Figure 13, P_1 processors are allocated to easy weight computation and P_2 processors to hard weight computation. Since weight vectors are computed for each pulse (Doppler bin), the parallelization in this step involves partitioning of data along dimension N , that is, each processor in easy weight computation task is responsible for $\frac{N_{easy}}{P_1}$ pulses while each processor in hard weight computation task is responsible for $\frac{N_{hard}}{P_2}$ pulses, as shown in Figure 16.

Notice that Doppler filter processing and weight computation tasks employ different data partitioning strategies (along different dimensions.) Due to different partitioning strategies, an all-to-all personalized communication scheme is required for data redistribution from Doppler filter processing task to the weight computation task. That is, each of the P_1 and P_2 processors needs to communicate with all P_0

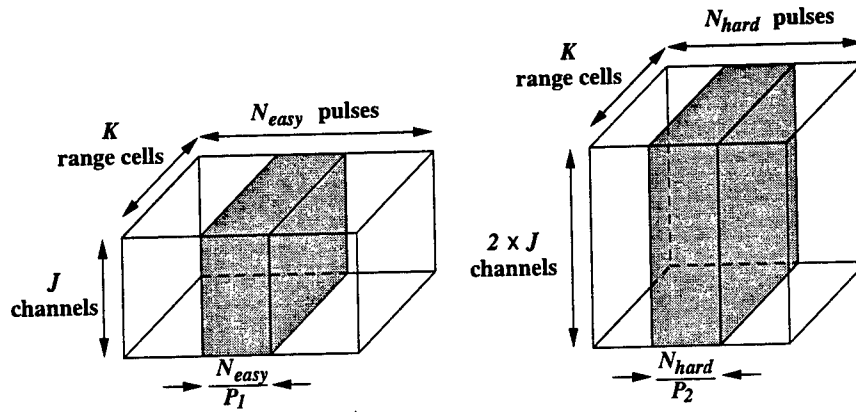


Figure 16. Partitioning strategy for easy and hard weight computation tasks. Data cube is partitioned across dimension N .

processors allocated to the Doppler filter processing task to receive CPI data. Since only subsets of Doppler filter processing task's output are used in the weight computation task, data collection is performed before inter-task communication. Although data collection reduces inter-task communication cost, it also involves data copying from non-continuous memory space to continuous buffers. Sometimes the cost of data collection may become extremely large due to hardware limitations (e.g. high cache miss ratio.) When sending data to the beamforming task, the weight vectors have already been partitioned along dimension N which is the same as the data partitioning strategy for the beamforming task. Therefore, no data collection is needed when transferring data to the beamforming task.

3.2.3 Beamforming

The third step in this pipeline (which is actually the second step for the current CPI because the result of the weight task is only used in the subsequent time step) is beamforming. The inputs of this task are received from both Doppler filter processing and weight computation tasks, as shown in Figure 13. The easy weight vector received from easy weight computation task is applied to the easy Doppler bins of

the received CPI data while the hard weight vector is applied to hard Doppler bins. The application of weights to CPI data requires matrix-matrix multiplications on two received data sets. Due to different matrix sizes for multiplications in easy and hard beamforming tasks, uneven computational load results. The beamforming task is also divided into easy and hard parts for parallelization purposes. This is because the easy and hard beamforming tasks require different amounts and portions of CPI data, and involve different computational loads. The inputs for the easy beamforming task are two 3-dimensional complex data cubes. One data cube which is received from the easy weight computation task is of size $N_{easy} \times M \times J$. The other is from Doppler filter processing task and its size is $N_{easy} \times J \times K$. A total of N_{easy} matrix-matrix multiplications are performed where each multiplication involves two matrices of size $M \times J$ and $J \times K$, respectively. The hard beamforming task also has two input data cubes which are received from Doppler filter processing and hard weight computation tasks. The data cube of size $6N_{hard} \times M \times 2J$ is received from hard weight computation task and the Doppler filtered CPI data cube is of size $N_{hard} \times 2J \times K$. Since range cells are divided into 6 range segments, there are a total of $6N_{hard}$ matrix-matrix multiplications in hard beamforming. The results of the beamforming task are two 3-dimensional complex data cubes of size $N_{easy} \times M \times K$ and $N_{hard} \times M \times K$ corresponding to easy and hard parts respectively.

In a manner similar to the weight computation task, parallelization in this step also involves partitioning of data across the N dimension (Doppler bins.) Different sets of processors are allocated to easy and hard beamforming tasks. Since the cost of matrix multiplications can be determined accurately, the computations are equally divided among the allocated processors for this task. As seen from Figure 13, this task requires data to be communicated from the first as well as the second task. Because data is partitioned along different dimensions, an all-to-all personalized communication is required for data redistribution between Doppler filter processing and beamforming tasks. The output of the Doppler filter processing task is a data cube of size $K \times 2J \times N$ which is redistributed to the beamforming task after data reorganization in the order of $N \times K \times 2J$. Data reorganization has to be done before the inter-task

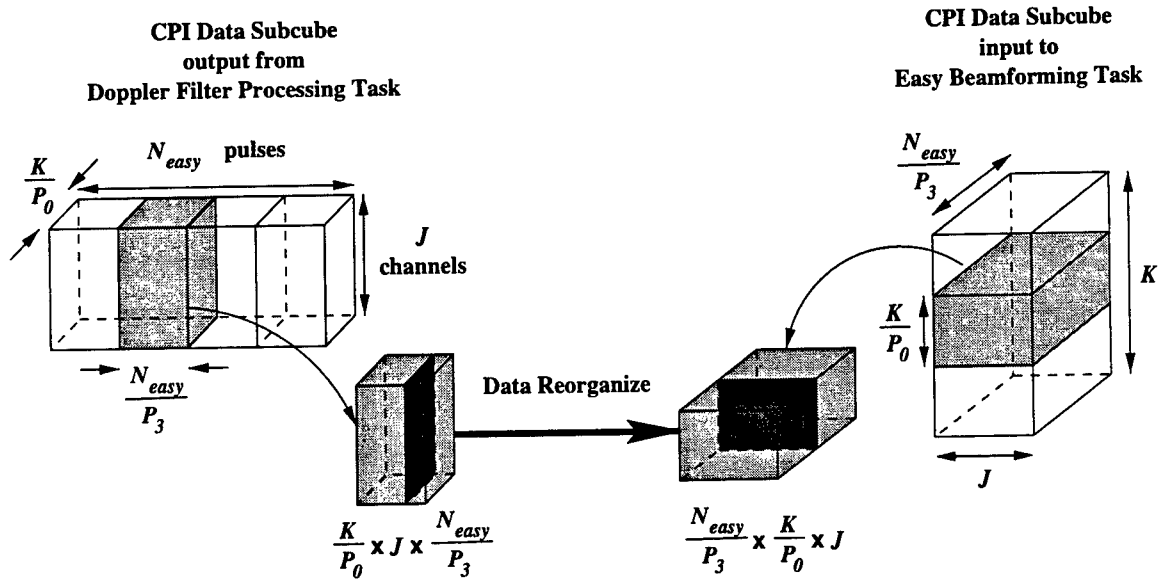


Figure 17. Data redistribution from Doppler filter processing task to easy beamforming task. CPI data subcube of size $\frac{K}{P_0} \times J \times \frac{N_{easy}}{P_3}$ is reorganized to subcube of size $\frac{N_{easy}}{P_3} \times \frac{K}{P_0} \times J$ before sending from one processor in Doppler filter processing task to another in easy beamforming task.

communication between the two tasks takes place, as shown in Figure 17.

Data reorganization involves data copying from non-continuous memory space and its cost may become extremely large due to cache misses. For example, two Doppler bins in the same range cell and the same channel are stored in contiguous memory space. After data reorganization, they are $\frac{K}{P_0} \cdot J$ element distance apart. Therefore, if P_0 is small and the size of CPI data subcube partitioned in each processor is large then it is quite likely that expensive data reorganization will be needed which becomes a major part of communication overhead. The algorithms which perform data collection and reorganization are crucial to exploit the available parallelism. Note that receiving data from weight computation tasks does not involve data reorganization or data collection because they have the same partitioning strategy (along dimension N .)

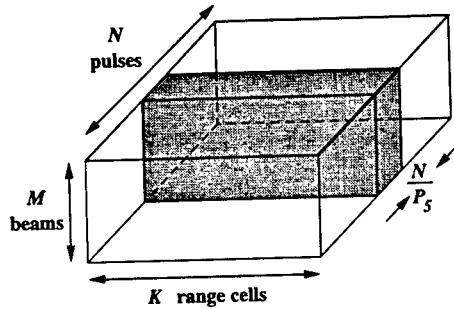


Figure 18. Partitioning strategy for pulse compression task. Data cube is partitioned across dimension N into P_5 processors.

3.2.4 Pulse Compression

The input to the pulse compression task is a 3-dimensional complex data cube of size $N \times M \times K$, as shown in Figure 18. This data cube consists of two subcubes of size $N_{easy} \times M \times K$ and $N_{hard} \times M \times K$ which are received from easy and hard beamforming tasks respectively. Pulse compression involves convolution of the received signal with a replica of the transmit pulse waveform. This is accomplished by first performing K -point FFTs on the two inputs, point-wise multiplication of the intermediate result and then computing the inverse FFT. The output of this step is a 3-dimensional real data cube of size $N \times M \times K$. The parallelization of this step is straightforward and involves the partitioning of data cube across the N dimension. Each of the FFTs could be performed on an individual processor and hence each processor in this task gets an equal amount of computation. Partitioning along the N dimension also results in an efficient accessing mechanism for continuous memory space when running FFTs. Since both beamforming and pulse compression tasks use the same data partitioning strategy (along dimension N), no data collection or reorganization is needed prior to communication between these two tasks. After pulse compression, the square of the magnitude of the complex data is computed to move to the real power domain. This cuts data set size in half and eliminates the computation of the square root.

3.2.5 CFAR Processing

The input to this task is an $N \times M \times K$ real data cube received from the pulse compression task. The sliding window constant false alarm rate (CFAR) processing compares the value of a test cell at a given range to the average of a set of reference cells around it times a probability of false alarm factor. This step involves summing up a number of range cells on each side of the cell under test, multiplying the sum by a constant, and comparing the product to the value of the cell under test. The output of this task, which appears at the pipeline output, is a list of targets at specified ranges, Doppler frequencies, and look directions. The parallelization strategy for this step is the same as for the pulse compression task. Both tasks partition data cube along the N dimension. Also, no data collection or reorganization is needed in pulse compression task before sending data to this task.

Figure 19 illustrates the organization of input/output data cubes for all tasks in this STAP algorithm. All data cubes shown in this figure are partitioned along its first dimension across the assigned processors in every task. For example, the input data cube to the Doppler filter processing task organized as $K \times J \times N$ is partitioned along dimension K .

3.3 Software Development and System Platform

All the parallel programs development and their integration was performed using ANSI C language and message passing interface (MPI) [7]. All the functions needed for data redistribution etc. were developed in the same fashion. This permits easy portability across various platforms which support C language and MPI. Since MPI is becoming a de facto standard for high-performance systems, we believe the software is portable. To facilitate upward or downward scalability, the number of processors, data sizes and other important parameters are runtime inputs so that the same program can be run on different number of processors without compiling it again. This allows, for example, the same function to be executed on 2, 4 and so on, number of processors.

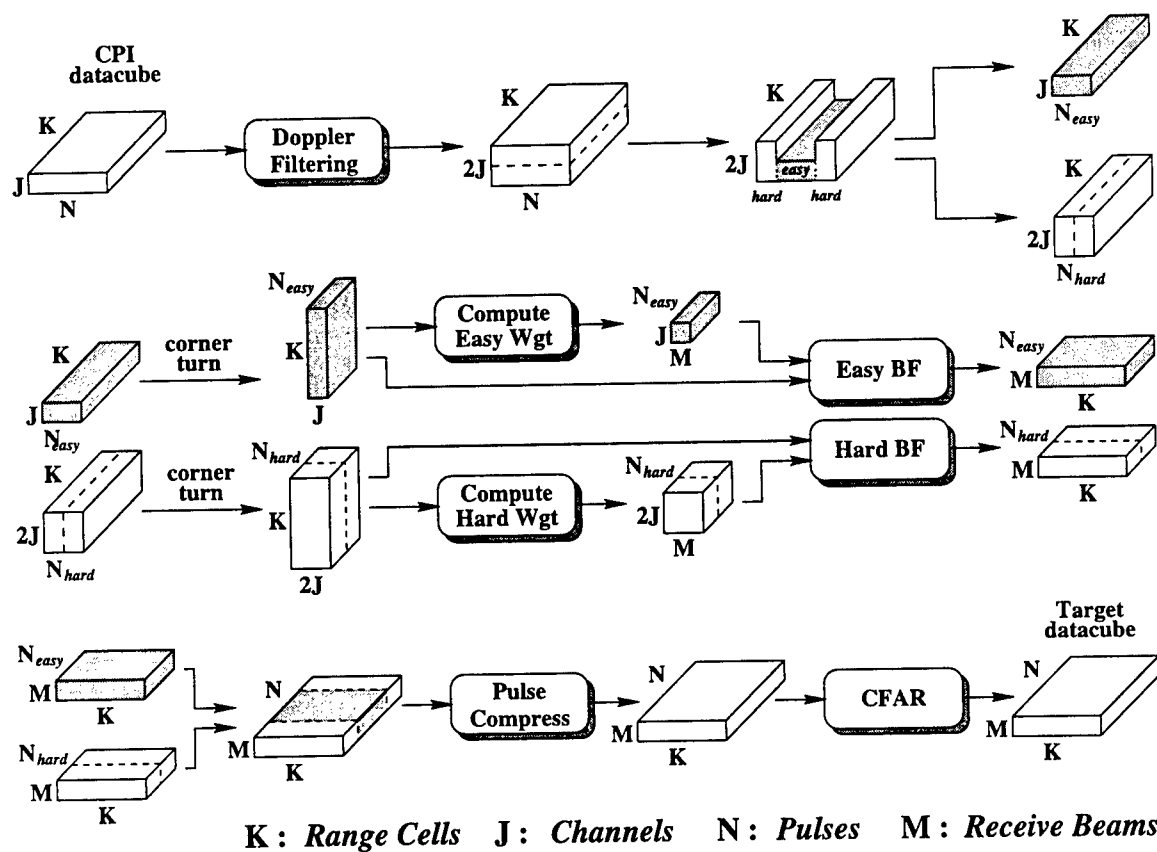


Figure 19. Organization of data cubes for all tasks in the STAP algorithm.

Table 3. Configurations of the system platforms on which we ran the parallel pipeline STAP codes.

	AFRL Paragon	ANL IBM SP	NWU SGI Origin
Configuration	MPP [†]	Cluster	Cluster
CPU Type	i860 RISC	P2SC [‡]	MIPS R10000
RAM (MByte)	64	256	1024
MFLOPS/proc	100	480	390
MHz /proc	40	120	195
No. nodes	232	80	8
No. proc/node	3	1	1
Execution mode	dedicated	dedicated	time shared

[†]MPP: Massively Parallel Processor

[‡]P2SC: Power 2 SuperScalar chip

The High Performance Computer (HPC) system we used to test our STAP code are: 232-node Intel Paragon at the Air Force Research Laboratory, 80-node IBM SP at Argonne National Laboratory, and 8-node SGI Origin at Northwestern University. Their system configurations are given in Table 3.

In our implementation, a double buffering strategy was used both in receive and send phases. During the execution loops, this strategy employs two buffers alternatively such that one buffer can be processed during the communication phase while the other buffer is processed during the compute phase. Together with the double buffering implementation, asynchronous send and receive calls were employed in order to maximize the overlap of communication and computation. The general execution flow and the approach to measure the timing for each part of computation and communication is given in Figure 20.

n : number of CPIs
inBuf[2] : input data buffer
outBuf[2] : output data buffer

```
1  for  $i \leftarrow 0$  to  $n - 1$ 
2     $prev \leftarrow (i - 1) \bmod 2$ 
3     $cur \leftarrow i \bmod 2$ 
4     $next \leftarrow (i + 1) \bmod 2$ 
5     $t_0 \leftarrow$  read timer
6    post async receives for inBuf[ $next$ ]
7    wait for completion of previous receives for inBuf[ $cur$ ]
8    data unpacking on inBuf[ $cur$ ]
9     $t_1 \leftarrow$  read timer
10   computation on inBuf[ $cur$ ] and result in outBuf[ $cur$ ]
11    $t_2 \leftarrow$  read timer
12   data packing for outgoing message on outBuf[ $cur$ ]
13   post async sends for outBuf[ $cur$ ] to next task
14   wait for completion of sends for outBuf[ $prev$ ]
15    $t_3 \leftarrow$  read timer
```

Figure 20. Implementation of timing computation and communication for each task. A double buffering strategy is used to overlap the communication with the computation. Receive time = $t_1 - t_0$, compute time = $t_2 - t_1$, and send time = $t_3 - t_2$.

Table 4. The number of floating point operations for the PRI-staggered post Doppler STAP algorithm to process one CPI data.

Task	number of floating point operations
Doppler filter processing	79,691,776
hard weight computation	197,038,464
easy weight computation	13,851,792
easy beamforming	28,311,552
hard beamforming	44,040,192
pulse compression	38,928,384
CFAR processing	1,690,368
Total	403,552,528

3.4 Performance Results

We specified the parameters that were used in our experiments as follows:

- range cells (K) = 512,
- channels (J) = 16,
- pulses (N) = 128,
- receive beams (M) = 6,
- easy Doppler bins (N_{easy}) = 72, and
- hard Doppler bins (N_{hard}) = 56.

Given these values of parameters, the total number of floating point operations (flops) required for each CPI data to be processed throughout this STAP algorithm is 403,552,528. Table 4 shows the number of flops required for each task. A total of 25 CPI complex data cubes were generated as inputs to the parallel pipeline system.

Each task in the pipeline contains three major parts: receiving data from the previous task, main computation, and sending results to the next task. Performance results are measured separately for these three parts, namely receiving time, computation time, and sending time. In each task timing results for processing one CPI data were obtained by accumulating the execution time for the middle 20 CPIs and then averaging it. Timing results presented in this chapter do not include the effect of initial setup (first 3 CPIs) and final iterations (last 2 CPIs).

3.4.1 Computation Costs

The task of computing hard weights is the most computationally demanding task. The Doppler filter processing task is the second most demanding task. Naturally, more processors are assigned to these two tasks in order to obtain a good performance. For each task in the STAP algorithm, parallelization was done by evenly dividing computational load across processors. Since there is no intra-task data dependency, no inter-processor communication occurs within any single task in the pipeline. Another way to view this is that intra-task communication is moved to the beginning of each task within the data redistribution step. Figure 21 gives the computation performance results as functions of numbers of processors and the corresponding speedup on the AFRL Intel Paragon. For each task, we obtained linear speedups.

3.4.2 Inter-task Communication

Inter-task communication refers to the communication between sending and receiving (distinct and parallel) tasks. This communication cost depends on both processor assignment for each task as well as on the volume and extent of data reorganization. Table 5 presents the inter-task communication timing results. Each sub-table considers pairs of tasks where the number of processors (# proc) for both tasks are varied. In some cases timing results shown in the tables contain idle time for waiting for the corresponding task to complete. This happens when receiving task's computation part completes before the sending task has generated data to send.

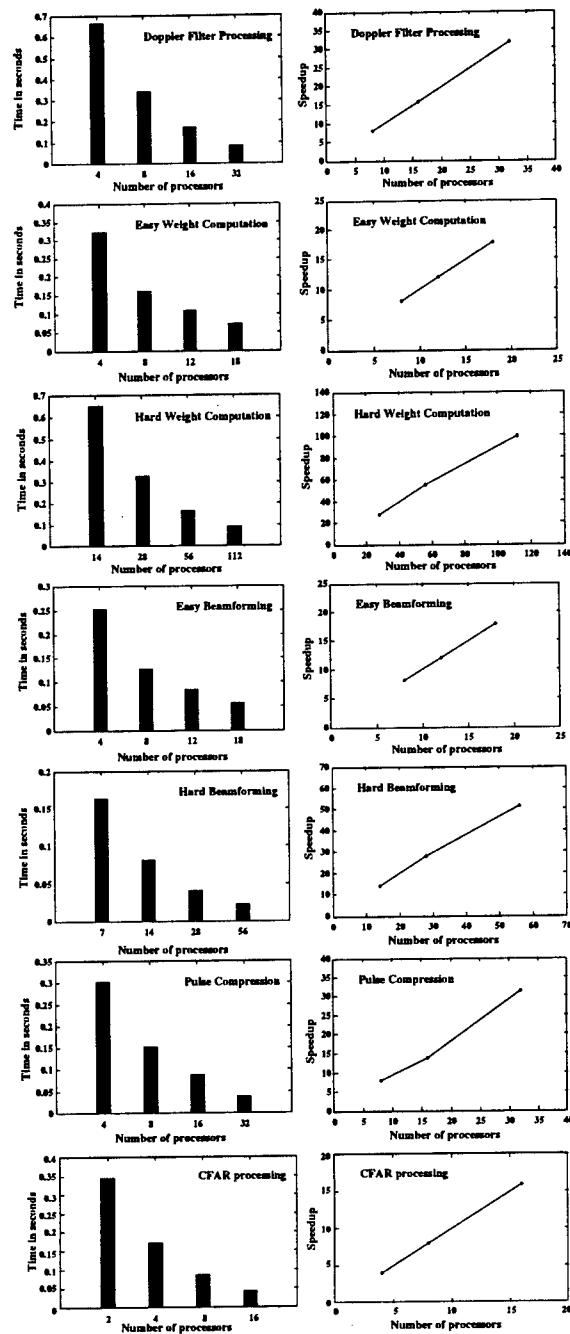


Figure 21. Performance and speedup of computation time as a function of number of processors for all tasks.

Table 5. Timing results of inter-task communication. Time in seconds. # proc: number of processors.

	# proc	easy weight		hard weight				easy BF		hard BF	
		16		56		112		16		16	
		send	recv	send	recv	send	recv	send	recv	send	recv
Doppler filter	8	.1332	.4339	.1332	.3603	.1332	.4441	.1332	.4509	.1332	.4395
	16	.0679	.1780	.0679	.1048	.0679	.1837	.0679	.1955	.0679	.1843
	32	.0340	.0511	.0332	.0034	.0340	.0563	.0340	.0646	.0340	.0519

	# proc	easy beamforming			
		8		16	
		send	recv	send	recv
easy weight	4	.0005	.1956	.0007	.2570
	8	.0088	.0883	.0004	.0905
	16	.0768	.0807	.0003	.0660

	# proc	hard beamforming			
		8		16	
		send	recv	send	recv
hard weight	28	.0007	.1798	.0007	.2485
	56	.0100	.1468	.0065	.0765
	112	.1824	.1398	.0005	.0543

	# proc	pulse compression			
		8		16	
		send	recv	send	recv
easy BF	4	.0069	.5016	.0069	.5714
	8	.0036	.1379	.0036	.2090
	16	.0580	.0771	.0022	.0569
hard BF	4	.0054	.5016	.0054	.5714
	8	.0029	.1379	.0030	.2090
	16	.1159	.0771	.0017	.0569

	#proc	CFAR processing			
		4		8	
		send	recv	send	recv
pulse compression	4	.0099	.3351	.0098	.3348
	8	.0053	.0662	.0051	.1750
	16	.1256	.0435	.0028	.1783

From most of the results the following important observations can be made. First, when the number of processors is unbalanced, the communication performance is not very good. Second, as the number of processors is increased in the sending and receiving tasks, communication scales tremendously. This happens for two reasons. One, each processor has less data to reorganize, pack and send and each processor has less data to receive; and two, contention at sending and receiving processors is reduced. Thus, it is not sufficient to improve the computation times for such parallel pipelined applications to improve throughput and latency.

In Figure 20 receiving time for each loop is given by subtracting t_1 from t_0 . Since

computation has to be performed only after input data has been received, receiving time may contain the waiting time for the input, shown in line 4. Sending time, $t_3 - t_2$, measures the time containing data packing (collection and reorganization) and posting sending requests. Because of the asynchronous send used in the implementation, the results shown here are visible sending time and the actual sending action may occur in other portions of the task. Similar to the receiving time, sending time may also contain waiting time for the completion of sending requests in the previous loop. With large number of processors, there is tremendous scaling in performance of communicating data as the number of processors is increased. This is because the amount of processing for communication per processor is decreased (as it handles less amount of data), amount of data per processor to be communicated is decreased and traffic on links going in and out of each processor is reduced. This model scales well for both computation and communication.

In Figure 20 receiving time for each loop is given by subtracting t_1 from t_0 . Since computation has to be performed only after input data has been received, receiving time may contain the waiting time for the input, shown in line 4. Sending time, $t_3 - t_2$, measures the time containing data packing (collection and reorganization) and posting sending requests. Because of the asynchronous send used in the implementation, the results shown here are visible sending time and the actual sending action may occur in other portions of the task. Similar to the receiving time, sending time may also contain waiting time for the completion of sending requests in the previous loop, shown in line 8. Especially in the cases when two communicating tasks have uneven partitioned parallel computation load, this effect becomes more apparent. With large number of processors, there is tremendous scaling in performance of communicating data as the number of processors is increased. This is because the amount of processing for communication per processor is decreased (as it handles less amount of data), amount of data per processor to be communicated is decreased and traffic on links going in and out of each processor is reduced. This model scales well for both computation and communication.

3.4.3 Integrated System Performance

Integrated system refers to the evaluation of performance when all the tasks are considered together. Throughput and latency are the two most important measures for performance evaluation in addition to individual task computation time and inter-task communication time. Table 6 gives timing results on the AFRL Paragon for three different cases with different processor assignments. The performance results on the ANL SP and SGI Origin at Northwestern University are given in Table 7.

In Section 3.2 equations (4) and (5) provide the throughput and latency for one CPI data set. The measured throughput is obtained by placing a timer at the end of last task and recording the time difference between every loop (that is between two successive completions of the pipeline.) The inverse of this measure provides the throughput. On the other hand, it is more difficult to measure latency because it requires synchronizing clocks at the first task and last task's processors. Thus, to obtain the measured latency, the timing measurement should be made by first reading time at both first task and last task when the first task is ready to read a new input data. This can be done by sending a signal from the first task to the last task's processor when the first task is ready for reading the new input data. Then the timer for last task can be started.

In fact, the latency given in equation (5) represents an **upper bound** because the way we time tasks contains the time of waiting for input from previous task. This waiting time portion overlaps with the computation time in the previous tasks and should be excluded from the latency. Thus the latency results are conservative values and the real latency is expected to be smaller than this value. However, the latency given from equation (5) indicates the worst-case performance for our implementation. The real latency equation, therefore, becomes

$$real \text{ latency} = T_0 + \max(T'_3, T'_4) + T'_5 + T'_6 \quad (6)$$

where $T'_i = T_i$ - idle time at receiving, $i = 3, 4, 5$, and 6 .

Figure 22 gives the throughput and latency results corresponding to the 3 cases in Table 6. From these 3 cases, it is clear that even for latency and throughput measures

Table 6. Performance results on the Intel Paragon for 3 cases of processor assignments.

Time in seconds. # proc: number of processors.

case 1: total number of processors = 236

	# proc	recv	comp	send	total
Doppler filter	32	.0055	.0874	.0348	.1276
easy weight	16	.0493	.0913	.0003	.1408
hard weight	112	.0555	.0831	.0005	.1390
easy BF	16	.0658	.0708	.0021	.1387
hard BF	28	.0936	.0414	.0010	.1361
pulse compr	16	.0551	.0776	.0028	.1355
CFAR	16	.0910	.0434	-	.1344
estimated		throughput		7.1019	
		latency		0.5362	
measured		throughput		7.2659	
		latency		0.3622	

case 2: total number of processors = 118

	# proc	recv	comp	send	total
Doppler filter	16	.0110	.1714	.0668	.2492
easy weight	8	.0998	.1636	.0003	.2637
hard weight	56	.0979	.1636	.0005	.2621
easy BF	8	.1302	.1267	.0036	.2605
hard BF	14	.1782	.0822	.0017	.2622
pulse compr	8	.1027	.1543	.0051	.2621
CFAR	8	.1742	.0864	-	.2606
estimated		throughput		3.7919	
		latency		1.0342	
measured		throughput		3.7959	
		latency		0.6805	

case 3: total number of processors = 59

	# proc	recv	comp	send	total
Doppler filter	8	.0219	.3509	.1296	.5024
easy weight	4	.1796	.3254	.0003	.5053
hard weight	28	.1779	.3265	.0006	.5050
easy BF	4	.2439	.2529	.0068	.5037
hard BF	7	.3370	.1636	.0032	.5039
pulse compr	4	.1806	.3067	.0097	.4970
CFAR	4	.3240	.1723	-	.4963
estimated		throughput		1.9791	
		latency		1.9996	
measured		throughput		1.9898	
		latency		1.3530	

Table 7. Performance results on IBM SP and SGI Origin.

IBM SP

case 1: total nodes = 52 Time in seconds

	node	recv	comp	send	total
Doppler	8	.0068	.0593	.0964	.1625
easy wgt	2	.1208	.0525	.0001	.1734
hard wgt	28	.1048	.0639	.0001	.1689
easy BF	4	.1072	.0605	.0001	.1678
hard BF	4	.1069	.0615	.0002	.1686
PC	4	.1146	.0527	.0001	.1674
CFAR	2	.1296	.0402	-	.1699
estimated	throughput		5.7654		
	latency		0.6684		
measured	throughput		5.9104		
	latency		0.4273		

IBM SP

case 3: total nodes = 8 Time in seconds

	node	recv	comp	send	total
Doppler	1	0.0484	.4240	.7688	1.2412
easy wgt	1	1.1360	.1051	.0001	1.2412
hard wgt	2	0.4950	.7464	.0001	1.2415
easy BF	1	1.0047	.2352	.0001	1.2399
hard BF	1	1.0018	.2387	.0001	1.2406
PC	1	1.0418	.1986	.0001	1.2405
CFAR	1	1.1602	.0802	-	1.2404
estimated	throughput		0.8055		
	latency		4.9627		
measured	throughput		0.8057		
	latency		2.5973		

IBM SP

case 2: total nodes = 26 Time in seconds

	node	recv	comp	send	total
Doppler	4	.0129	.1070	.2031	.3230
easy wgt	1	.2230	.1021	.0001	.3252
hard wgt	14	.2182	.1072	.0001	.3255
easy BF	2	.2052	.1185	.0001	.3238
hard BF	2	.2054	.1189	.0001	.3244
PC	2	.2231	.0989	.0001	.3221
CFAR	1	.2439	.0809	-	.3248
estimated	throughput		3.0179		
	latency		1.2942		
measured	throughput		3.0810		
	latency		0.9062		

SGI Origin

case 4: total nodes = 8 Time in seconds

	node	recv	comp	send	total
Doppler	1	0.0695	.6437	.8540	1.5671
easy wgt	1	1.4808	.0924	.0000	1.5732
hard wgt	2	0.9531	.6208	.0005	1.5744
easy BF	1	1.3615	.2001	.0005	1.5620
hard BF	1	1.3395	.2220	.0012	1.5627
PC	1	1.3973	.1638	.0018	1.5628
CFAR	1	1.5326	.0303	-	1.5629
estimated	throughput		0.6352		
	latency		6.2554		
measured	throughput		0.6395		
	latency		3.0983		

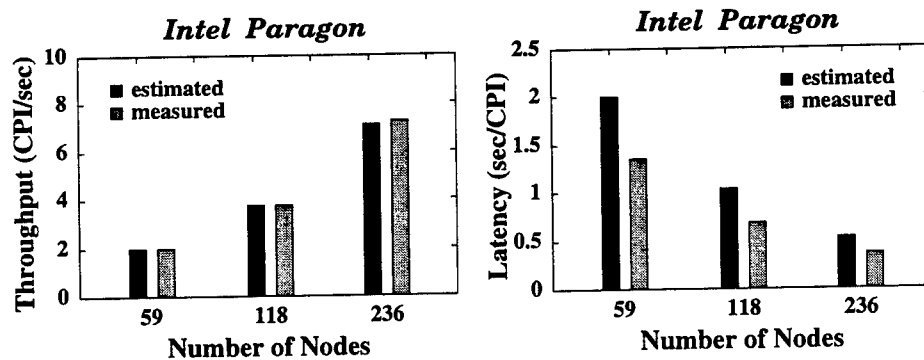


Figure 22. Throughput and latency for the 3 cases in Table 6. Measured results are obtained from the experiments while estimated results are obtained from applying individual tasks' timing to equations (4) and (5). The unit of throughput is number of CPIs per second. The unit of latency is second.

we obtain linear speedups from our experiments. Figure 23 shows the performance results corresponding to Table 23. We were limited to these number of processors due to the size of the machines. Both throughput and latency results scale well on the IBM SP at ANL. Given that this scale up is up to 236 processors on the Paragon and 52 processors on the SP, we believe these are very good results.

As discussed in Chapter 2, tradeoffs exist between assigning processors to maximize throughput and to minimize latency, given limited resources. Using two examples, we illustrate how further performance improvements may (or may not) be achieved if few extra processors are available. We now take case 2 from Table 6 as an example and add some extra processors to tasks to analyze its effect to the throughput and latency. Suppose that case 2 has fulfilled the minimum throughput requirement and more processors can be added. Table 8 shows that adding 4 more processors to Doppler filter processing task not only increases the throughput but also reduces the latency. This is because the communication amount for each send and receive between Doppler filter processing task to weight computation and to beamforming tasks is reduced (Table 8). So, clearly adding processors to one task not only affects that

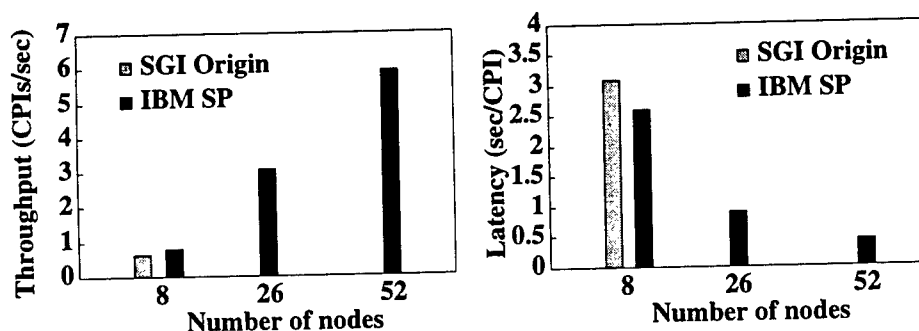


Figure 23. Throughput and latency results correspond to the cases in Table 7.

task's performance but has a measurable effect on the performance of other tasks. By increasing the number of processors 3%, the improvement in throughput is 32% and in latency is 19%. *Such effects are very difficult to capture in purely theoretical models because of the secondary effects.*

Since the parallel computation load may be different among tasks, bottleneck problems arise when some tasks in the pipeline do not have proper numbers of processors assigned. If the number of processors assigned to one task with heavy work load is not enough to catch up the input data rate, this task becomes a bottleneck in the pipeline system. Hence, it is important to maintain approximately the same computation time among tasks in the pipeline system to maximize the throughput and also achieve higher processor utilization. One bottleneck task can be seen when its computation time is relatively much larger than the rest of the tasks. The entire system's performance degrades because the rest of the tasks have to wait for bottleneck task's completion to send/receive data to/from it no matter how many more processors assigned to them and how fast they can complete their jobs. Therefore, poor task scheduling and processor assignment will cause significant portion of idle time in the resulted communication costs. In Table 9 we added a total of 16 more processors to pulse compression and CFAR processing tasks to the case in Table 8. Comparing to case 2 in Table 6, we can see that the throughput increased. However, the throughput did not improve compared to the results in Table 8, even though this assignment has 16 more processors. In this case, the weight tasks are bottleneck tasks

Table 8. Performance results for adding 4 more processors to Doppler filter processing task to case 2 in Table 6. Time in seconds.

total number of processors = 122					
	# proc	recv	comp	send	total
Doppler filter	20	.0090	.1395	.0540	.2024
easy weight	8	.0519	.1633	.0003	.2155
hard weight	56	.0486	.1644	.0005	.2135
easy BF	8	.0815	.1272	.0037	.2124
hard BF	14	.1232	.0823	.0018	.2073
pulse compr	8	.0519	.1543	.0051	.2113
CFAR	8	.1240	.0864	-	.2105
throughput	5.0213				
latency	0.5498				

because their computation costs are relatively higher than other tasks. We can see that the receiving time of the rest of tasks are much larger than their computation time. A significant portion of idle time waiting for the completion of weight tasks is in the receiving time. On the other hand, we observe 23% improvement in the latency. This is because the computation time is reduced in the last two tasks with more processors assigned. From equation (6), the execution time of these two tasks, T'_5 and T'_6 , decreases and therefore the latency is reduced.

3.5 Summary

In this chapter we presented the design and implementation for a PRI-staggered post-Doppler STAP algorithm implementation on the Intel Paragon machine at AFRL, the IBM SP at ANL and SGI Origin at Northwestern University. The performance results indicate that our approach of parallel pipelined implementation scales well both in terms of communication and computation. For the integrated pipeline system, the throughput and latency also demonstrate the linear scalability of our design. Our

Table 9. Performance results for adding 16 more processors to pulse compression and CFAR processing tasks to the case in Table 8. Time in seconds.

total number of processors = 138

	# proc	recv	comp	send	total
Doppler filter	20	.0091	.1395	.0541	.2027
easy weight	8	.0516	.1633	.0003	.2152
hard weight	56	.0488	.1644	.0005	.2137
easy BF	8	.0819	.1273	.0037	.2129
hard BF	14	.1301	.0823	.0018	.2142
pulse compr	16	.1337	.0775	.0028	.2140
CFAR	16	.1701	.0434	-	.2135
throughput	4.9052				
latency	0.4247				

design and implementation not only shows tradeoffs in parallelization, processor assignment, and various overheads in inter and intra-task communication etc., but it also shows that accurate performance measurement of these systems is very important. Consideration of issues such as cache performance when data is packed and unpacked, and impact of the parallelization and processor assignment for one task on another task are crucial. This is normally not easily captured in theoretical models.

Chapter 4

Multi-Threaded Design and Implementation

In this chapter, we present the multi-threaded design and implementation for the parallel pipelined STAP system on Intel Paragon MP system. The Intel Paragon at the Air Force Research Laboratory (AFRL), Rome, New York, is an MP system which has three processors on each compute node board. By running UNIX OSF/1 operating system, each node can run multiple processes and each process can have multiple threads at the same time. In this chapter, we focus on the design of the parallel pipeline system and its implementation using multi-threading on this system. Our goal is to determine the performance enhancement that can be achieved when using small SMPs on each node of a large parallel computer for such an application. We also discuss the process of software development for such an application on parallel computers when latency and throughput are both considered together and present their tradeoffs. We demonstrate the performance improvement and scalability on different numbers of compute nodes for both threaded and non-threaded implementations. The performance improvement results for the threaded implementation over non-threaded implementation are provided. Due to limitations of software in the Intel Paragon, the improvement is not as good as expected on the system with multi-processors on each compute node board.

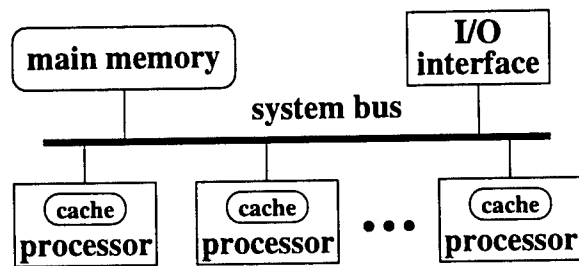


Figure 24. The architecture of a Symmetrical Multi-Processor system.

The rest of the chapter is organized as follows: Section 4.1 describes the architecture of symmetrical multi-processor systems and the multi-threaded programming environment on the Intel Paragon MP system. Section 4.2 presents the multi-threaded design and implementation of the parallel pipeline STAP system. Performance results are given in Section 4.3.

4.1 Symmetrical Multi-Processor System

Symmetrical Multi-Processor (SMP) systems were introduced during the 1960s for mainframe computers. The concept of multiprogramming was first introduced on uni-processors with the goal of providing scaleup by overlapping CPU and I/O times and to support the time sharing of system resources by numerous users. Figure 24 illustrates the architecture of an SMP system. In an SMP system, there are multiple processors each having its own private cache memory and having an equal access to the other system resources such as the main memory and I/O. The SMP architecture is favored in the 1990s because it is the most affordable way to achieve scalability; i.e., just plugging in one processor board provides an increase in performance.

The development of the SMP programming environment was based on the fact that the main memory is common and is accessible to all processors running in the system. With the introduction of threads, or lightweight processes, the basic concept of multiprogramming is to allow more than one execution stream to work on the same workload. Each thread is an independent execution stream that synchronizes

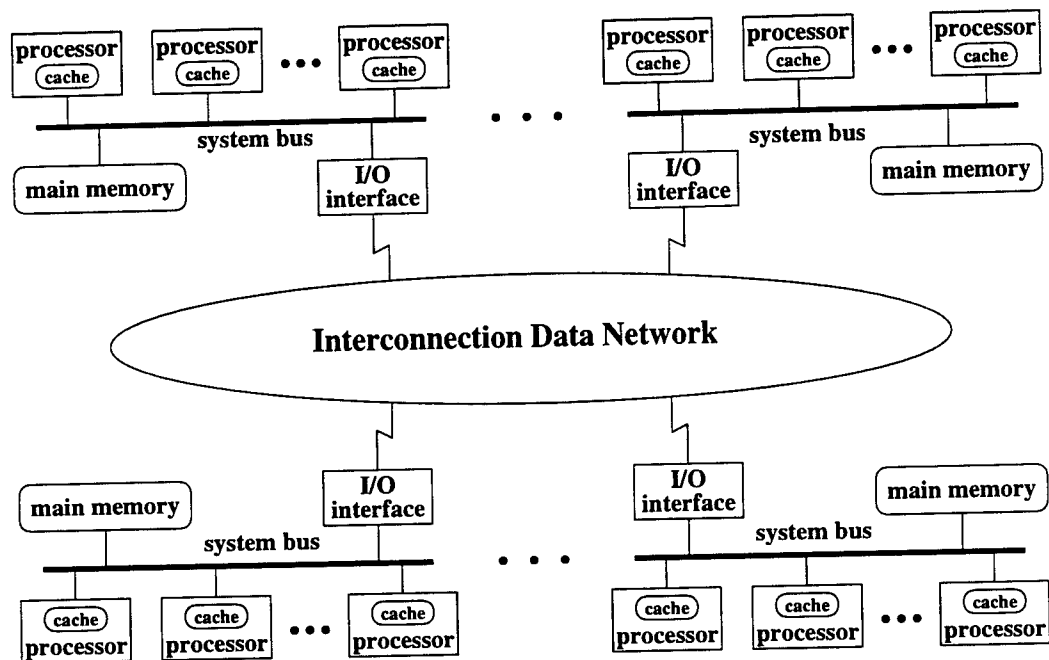


Figure 25. The architecture of an Massively Parallel Processing system with SMP nodes.

its accesses to common data in the main memory with other threads by using locks to prevent simultaneous updating. The operating systems running on the SMP systems must have proper scheduling algorithms to evenly distribute all threads among available processors. In this way, multiple threads with the same copy of binary code can be executed concurrently on more than one CPU and, therefore, the SMP system scalability is achieved.

The Massively Parallel Processing (MPP) computers with SMP nodes are configured with a large set of SMP nodes linked by high speed interconnection data network. Processor communication within the a SMP node is carried out by accessing shared main memory. Processors in different SMP nodes communicate with each other using message passing through the interconnection data network. The architecture of an MPP system with SMP nodes is given in Figure 25. An Intel Paragon MP system is an example of this type of architecture.

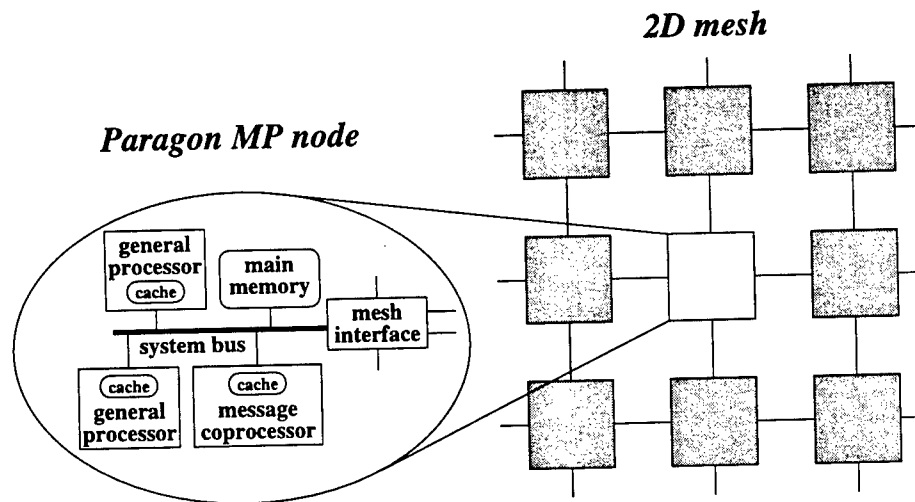


Figure 26. The architecture of an Intel Paragon MP system.

4.1.1 Intel Paragon MP System

We implemented our parallel pipeline model of the STAP algorithm on the Intel Paragon XP/S parallel computer located at Air Force Research Laboratory (AFRL) in Rome, New York. The compute partition of this machine consists of 307 MP nodes, each with 64M byte RAM. All 307 MP nodes are connected by a high-speed node interconnect network and are configured in a two-dimensional mesh. Of the 307 MP nodes, 232 are general compute nodes which run users' applications. Every Paragon MP node is a SMP system with three i860 processors on each compute node board. The architecture of a Paragon MP node is illustrated in Figure 26. Each of the three processors has its own private cache memory but shares the main memory with the other two processors. The operating system is a version of UNIX OSF/1. By running this operating system, the three processors in each compute node are configured with two processors as general application processors and one processor as message coprocessor which is dedicated to message passing. Multi-threaded programming environment is supported on a Paragon system [29]. The threads are implemented as *POSIX threads* which are based on the *POSIX Threads Extension [C language] P1003.4a/D4 (Draft 4), August 1990*. Therefore, the programs that use

POSIX threads may not be portable to other systems.

Since two out of the three processors in the Paragon MP system are configured as general application processors, threads in a multi-threaded program on the MP system can run on either of the two application processors. Each thread runs independently, but shares resources with other threads. For example, all the threads in a single process share the main memory. Each compute node acts just like a parallel shared memory system with two processors. Ideally, if multi-threaded programs have no concurrent write operations, a speedup of 2 can be expected by using threads on a compute node of the Paragon MP system.

4.2 Design and Implementation

The STAP algorithm we implemented is described in Chapter 3. The structure of the parallel pipelined STAP system is the same as shown in Figure 13. From a single task point of view, the execution flow consists of three phases: receive, compute, and send phases, shown in Figure 8. In this chapter, only the compute phase is to re-designed so as to embed multiple threads.

4.2.1 Threads in the Compute Phase

The Intel Paragon at the AFRL is an MP system which has three processors on each compute node board. In each compute node, two out of the three processors are configured as general processors to run application code while the third is a message coprocessor which is dedicated to message passing. With this configuration, only the compute phase for each task in our parallel pipeline system is implemented with threads. The reason for not implementing threads in the communication phase is that the Paragon message-passing library is not thread-safe. Also, if more than one thread performs message passing, the message-passing performance may degrade and results may be incorrect. The message passing thread can be the main thread or any other thread. However, a thread other than the main thread will experience higher message

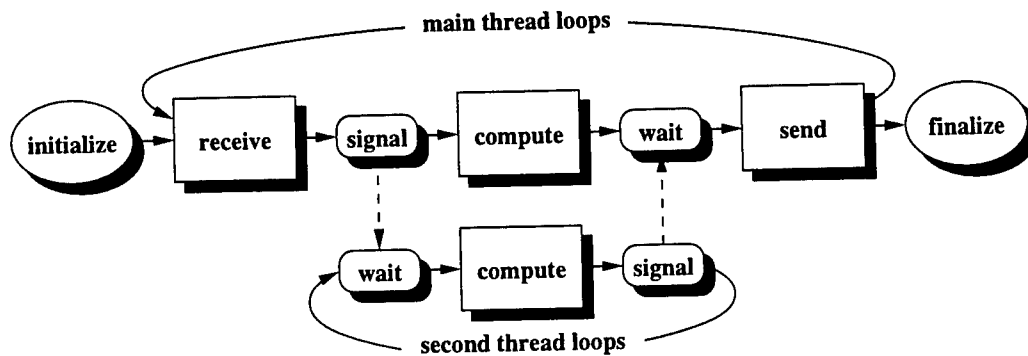


Figure 27. Implementation of two threads in the compute phase. The main thread signals the second thread to perform its computation. After completion of its computation, the second thread signals back to the main thread.

latency than the main thread. Besides, one processor has already been configured as message coprocessor which is dedicated to message passing and the communication performance has been sufficiently improved on the Paragon system.

Since there are only two application processors in each compute node, each compute phase in every task will have two threads implemented. For each task, the main thread in the compute phase sends a signal to the second thread when the input data is ready at the receive phase. Both threads then perform the computation on two processors concurrently. Once the second thread completes its computation, it signals the main thread that its output data is ready so that the main thread can start the send phase. While the main thread is performing the message passing calls, the second thread is waiting for its input signal from the main thread. These two signal operations involve two synchronizations of two threads using a mutually exclusive access semaphore. Figure 27 gives the execution flows of two threads in the compute phase.

4.2.2 Software Development

All the parallel program development and their integration was performed using ANSI C language. The libraries linked include standard C math library, message passing interface (MPI) library [7], POSIX thread library, and Kuck and Associates' CLASSPACK basic math library [30]. CLASSPACK library includes several basic linear algebra subroutines (BLAS) and Fast Fourier transform subroutines. The BLAS contains useful vector and matrix operations for dense numerical linear algebra programs. All subroutines in CLASSPACK library have been tuned for optimal performance on the Intel Paragon. The thread-safe versions of these libraries are also provided in the Intel Paragon and linked by the multi-threaded version of parallel pipeline STAP implementation.

In our implementation, a double buffering strategy was used both in receive and send phases. During the execution loops, this strategy employs two buffers alternately such that one buffer can be processed during the communication phase while the other buffer is processed during the compute phase. Together with the double buffering implementation, asynchronous send and receive calls were employed in order to maximize the overlap of communication and computation. The general execution flow and the approach to measure the timing for each part of computation and communication is given in Figure 20.

4.3 Performance Results

The implementation of the STAP application based on our parallel pipeline system model was done on the Intel Paragon at the Air Force Research Laboratory, Rome, New York. Each CPI complex data cube is a $512 \times 16 \times 128$ three-dimensional array. A total of 25 CPIs were generated as inputs to the parallel pipeline system. In each task, timing results for processing one CPI data cube were obtained by accumulating the execution time for the middle 20 CPIs and then averaging it. Timing results presented in this chapter do not include the effect of initial setup (first 3 CPIs) and

final stage (last 2 CPIs). Each task in the pipeline contains three parts: receiving data from the previous task, main computation, and sending results to the next task. Performance results are measured separately for these three parts, namely receive time, compute time, and send time. Since the multiple thread strategy is implemented in the compute phase only, we first discuss the compute time for each task in the pipeline and then present the performance results for the integrated pipeline system.

4.3.1 Compute Time

The task of computing hard weights is the most computationally demanding task. The Doppler filter processing task is the second most demanding task. Naturally, more compute nodes are assigned to these two tasks in order to obtain a good performance. For each task in the STAP algorithm, parallelization was done by evenly dividing computational load across compute nodes assigned. Figure 28 gives the performance results for different tasks during the compute phase on the AFRL Intel Paragon. It includes the execution time, the corresponding speedup, and the threading speedups when using two threads over a non-threaded implementation, all as functions of numbers of compute nodes. For each task, we obtained linear speedups for both two threads and single thread implementations. From Figure 28(b), the speedups when using two threads are approximately the same as using a single thread.

Assuming that the execution time of a non-threaded implementation of a task is t_1 and the execution time of its threaded implementation is t_2 , we define the threading speedup for threaded over non-threaded implementation as

$$s = \frac{t_1}{t_2}. \quad (7)$$

Since two processors are employed in the threaded implementation, we have $\frac{t_1}{2} \leq t_2 \leq t_1$ and therefore $1 \leq s \leq 2$. The threading speedups for all the tasks during the compute phase are given in Figure 28(c). By running on two processors at the same time, the two threaded STAP code ideally can have a threading speedup of 2. However, in most cases, the actual threading speedups do not approach this ideal value. This may

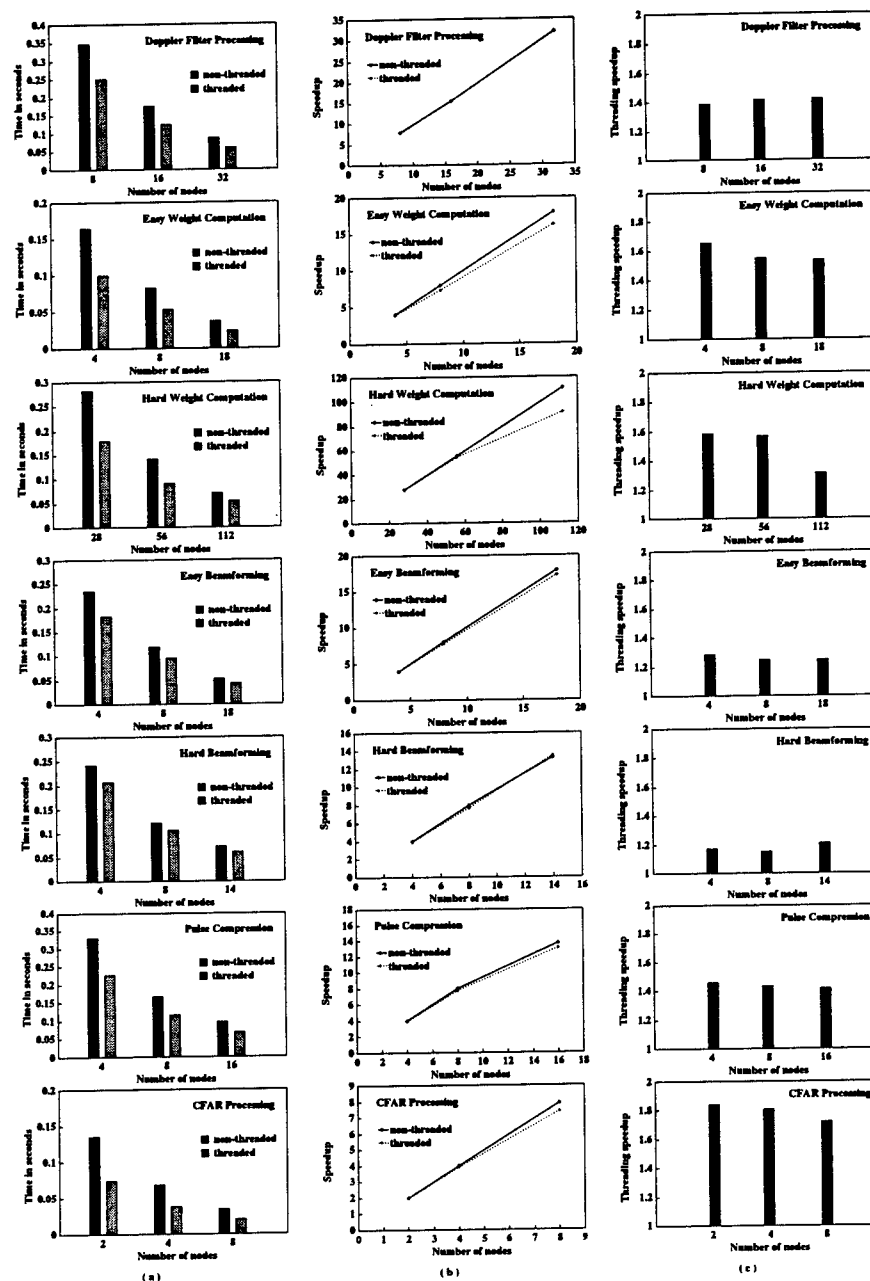


Figure 28. Performance of different tasks during the compute phase as a function of the number of compute nodes: (a) execution time, (b) speedups, and (c) threading speedups.

be caused by the limitation of implementation of the operating system, OSF/1, and the implementation of linked thread-safe libraries. On an Intel Paragon MP system, scheduling of threads is handled by the operating system kernel. Users do not have control over or get information about which processor runs which thread. On the other hand, the implementation of thread-safe versions of linked libraries most likely contains overheads of concurrent read/write operations when multiple threads are taken into consideration. Although each thread in a process executes independently, it shares resources with other threads, for example, the memory. Concurrent read and write operations prevent the threaded implementation from obtaining a linear speedup, even if two processors are used concurrently.

4.3.2 Integrated System Performance Evaluation

Integrated system performance evaluation refers to the evaluation of performance when all the tasks in the pipeline are considered together. Throughput (number of CPIs per second) and latency (number of seconds per CPI) are the two most important measures for performance evaluation on the parallel pipeline system. Tables 10 and 11 provides detailed timing results for three cases of different compute node assignments, each with threaded and non-threaded implementations. These timing tables include computation time and communication time of each task for processing one CPI. Because of the asynchronous send and receive calls used in the implementation, the results shown here are communication times that can actually be measured.

Figure 29 gives the estimated and measured throughput and latency values corresponding to Tables 10 and 11. Given timing results for each individual task, estimated throughput and latency are obtained by applying these individual timing results to Equations (4) and (5), shown in Section 3.2 of Chapter 3. The measured throughput and latency are obtained by placing a timer at the end of the last task and recording the time difference between every loop (that is, between two successive completions of the pipeline.) The measured throughput results are very close to the estimated ones both for threaded and non-threaded implementations. However, the measured

Table 10. Performance results of non-threaded implementation for 3 cases of nodes assignments.

case 1: total number of nodes = 176				Time in seconds	
	# nodes	recv	comp	send	total
Doppler filter	32	.0052	.0860	.0344	.1256
easy weight	8	.0482	.0824	.0004	.1310
hard weight	84	.0373	.0947	.0003	.1323
easy BF	18	.0752	.0561	.0002	.1315
hard BF	14	.0547	.0696	.0002	.1246
pulse compr	16	.0364	.0834	.0085	.1284
CFAR	4	.0597	.0677	-	.1273
estimated			throughput	7.5579	
			latency	0.5128	
measured			throughput	7.7403	
			latency	0.3985	
case 2: total number of nodes = 102				Time in seconds	
	# nodes	recv	comp	send	total
Doppler filter	16	.0102	.1761	.0701	.2563
easy weight	4	.0957	.1640	.0003	.2600
hard weight	56	.1184	.1410	.0003	.2597
easy BF	8	.1409	.1178	.0003	.2590
hard BF	8	.1335	.1214	.0003	.2551
pulse compr	8	.0746	.1653	.0150	.2548
CFAR	2	.1199	.1351	-	.2550
estimated			throughput	3.8460	
			latency	1.0251	
measured			throughput	3.8677	
			latency	0.7767	
case 3: total number of nodes = 51				Time in seconds	
	# nodes	recv	comp	send	total
Doppler filter	8	.0193	.3471	.1364	.5028
easy weight	2	.1827	.3273	.0003	.5102
hard weight	28	.2293	.2815	.0003	.5110
easy BF	4	.2715	.2347	.0003	.5065
hard BF	4	.2538	.2423	.0002	.4963
pulse compr	4	.1359	.3297	.0293	.4949
CFAR	1	.2256	.2695	.0000	.4950
estimated			throughput	1.9569	
			latency	1.9992	
measured			throughput	1.9962	
			latency	1.5151	

Table 11. Performance results of threaded implementation for 3 cases of nodes assignments.

case 1: total number of nodes = 176					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	32	.0052	.0618	.0369	.1039
easy weight	8	.0605	.0538	.0004	.1146
hard weight	84	.0518	.0615	.0004	.1137
easy BF	18	.0680	.0439	.0004	.1123
hard BF	14	.0447	.0602	.0004	.1054
pulse compr	16	.0399	.0608	.0084	.1091
CFAR	4	.0701	.0376	-	.1076
estimated throughput				8.7243	
latency				0.4329	
measured throughput				9.1895	
latency				0.3248	
case 2: total number of nodes = 102					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	16	.0101	.1246	.0739	.2086
easy weight	4	.1190	.0992	.0004	.2185
hard weight	56	.1238	.0909	.0004	.2151
easy BF	8	.1223	.0946	.0002	.2171
hard BF	8	.1035	.1056	.0003	.2094
pulse compr	8	.0805	.1151	.0153	.2109
CFAR	2	.1357	.0735	-	.2091
estimated throughput				4.5757	
latency				0.8457	
measured throughput				4.6916	
latency				0.6108	
case 3: total number of nodes = 51					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	8	.0201	.2502	.1429	.4132
easy weight	2	.2241	.1939	.0003	.4183
hard weight	28	.2388	.1777	.0004	.4169
easy BF	4	.2301	.1832	.0004	.4136
hard BF	4	.1935	.2070	.0003	.4009
pulse compr	4	.1447	.2262	.0298	.4006
CFAR	1	.2546	.1451	-	.3997
estimated throughput				2.3905	
latency				1.6272	
measured throughput				2.4590	
latency				1.2046	

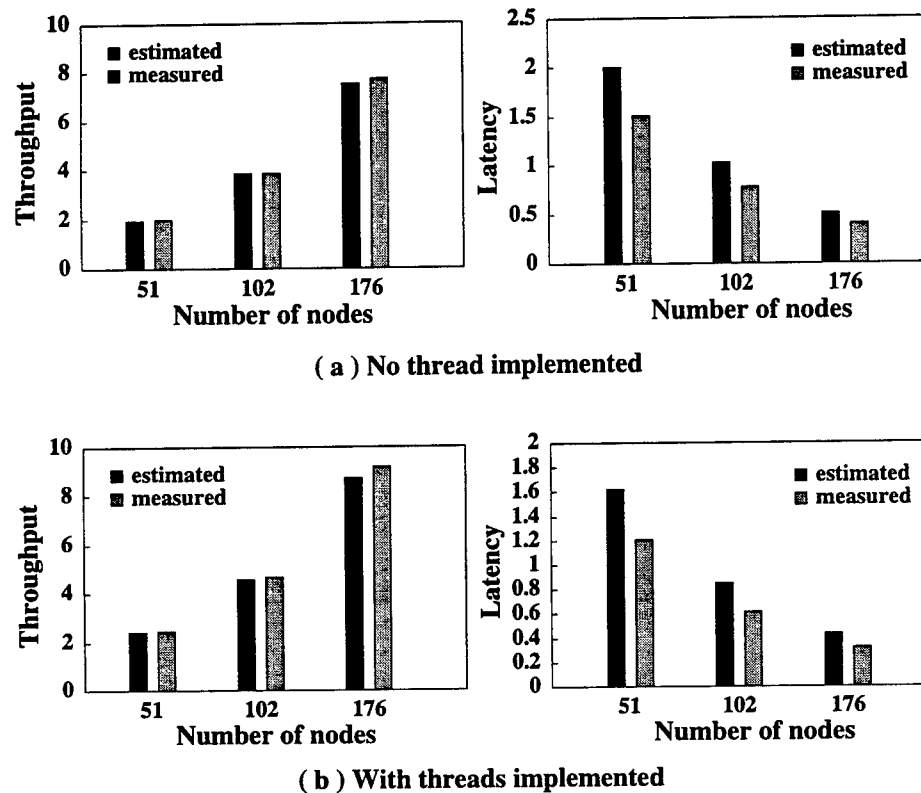


Figure 29. Estimated and measured values of throughput (number of CPIs per second) and latency (seconds per CPI) for both threaded and non-threaded implementations.

latency results are smaller than the estimated ones. It is because some tasks may need to wait for their input data from the previous tasks and this waiting time actually overlaps with the computation time of the previous tasks. This waiting time should be excluded from the actual latency value. The latency obtained from Equation (5) yields the worst-case performance for our implementation. The real latency is expected to be smaller than the estimated value.

Figure 30 shows the speedups and threading speedups achieved by the threaded implementation for both latency and throughput corresponding to three cases of compute node assignments. From these experiments, it is clear that for latency and throughput measures we obtain linear speedups for both threaded and non-threaded

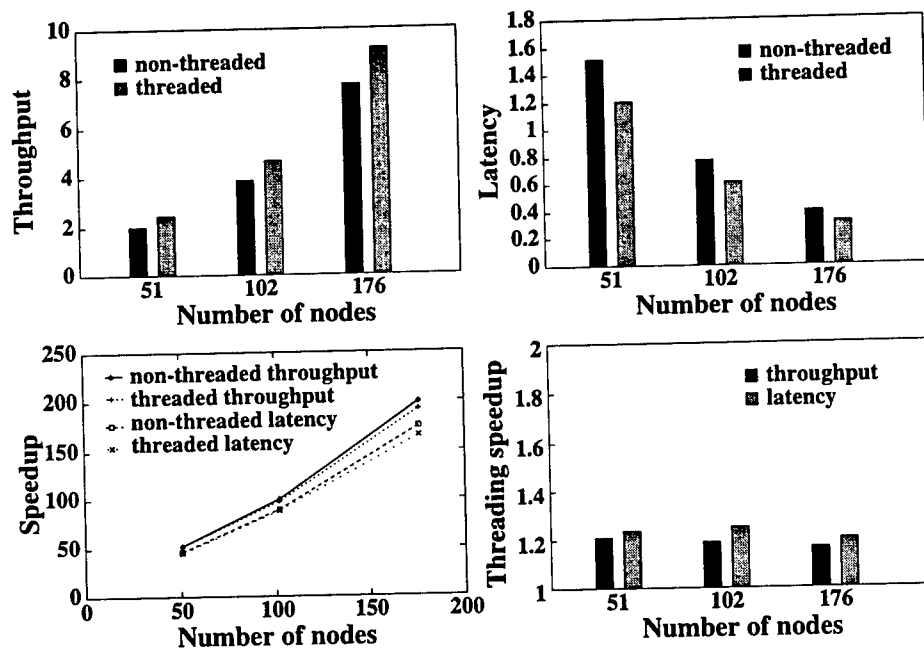


Figure 30. Performance results of integrated pipeline system for threaded and non-threaded implementations, corresponding to Tables 10 and 11.

implementations. Given that this scale up is up to 176 compute nodes (we were limited to this number of nodes due to the size of the machine), we believe these are very good results.

4.3.3 Tradeoff Between Throughput and Latency

As discussed in Chapter 2, tradeoffs exist between the assignment of compute nodes to maximize the overall throughput and the assignment of compute nodes to minimize latency, given limited resources. Using an example, we illustrate how further performance improvements may (or may not) be achieved if a few extra compute nodes are available. We now take the case with 102 nodes from Tables 10 and 11 as an example and add some extra compute nodes to the pipeline to analyze its effect on the throughput and latency. Extra compute nodes were added to each task in

increments of two nodes at a time. The resulting throughput and latency are plotted in Figure 31.

When extra compute nodes were added to Doppler filter processing tasks, the throughput increased and latency reduced. From Equations (4) and (5), this improvement was obtained because the execution time, T_0 , is reduced. However, when the number of nodes added is more than 8, both throughput and latency degrade. This is because the Doppler filter processing task finishes its computation on the new CPI so fast that the actual sending operations for the previous CPI have not been carried out yet. The Doppler filter processing task is forced to wait until the previous send operations complete. At this moment, the clock has already been read for the new CPI to be used later to calculate the throughput and latency. The waiting time increases Doppler filter processing task's execution time, T_0 , and therefore degrades the throughput and latency.

When compute nodes are added to easy and hard weight computation tasks, the resulting throughput and latency have no significant changes. This is because the latency does not contain the execution time of weight computations, as indicated in Equation (5). In the case with 102 nodes, we observe that the Doppler filter processing task has the maximum execution time among all tasks. From Equation (5), the throughput is affected only by the execution time of Doppler filter processing task. Therefore, further reduction of the execution time for weight computations does not improve the throughput.

However, when extra compute nodes are added to either the beamforming or the pulse compression task, we observe that the latency is reduced. This is because the execution times T_3 , T_4 , and T_5 reduce in Equation (5). The throughput, on the other hand, is still not improved because the Doppler filter processing task is still the task with the maximum execution time among all tasks.

Given additional compute nodes, Figure 31 presents the tradeoffs between increasing the throughput and reducing the latency, when assigning nodes to the tasks in the pipeline. Let us consider the case with 102 compute nodes in Tables 10 and 11 that has satisfied the maximum response time requirement (latency) and more compute

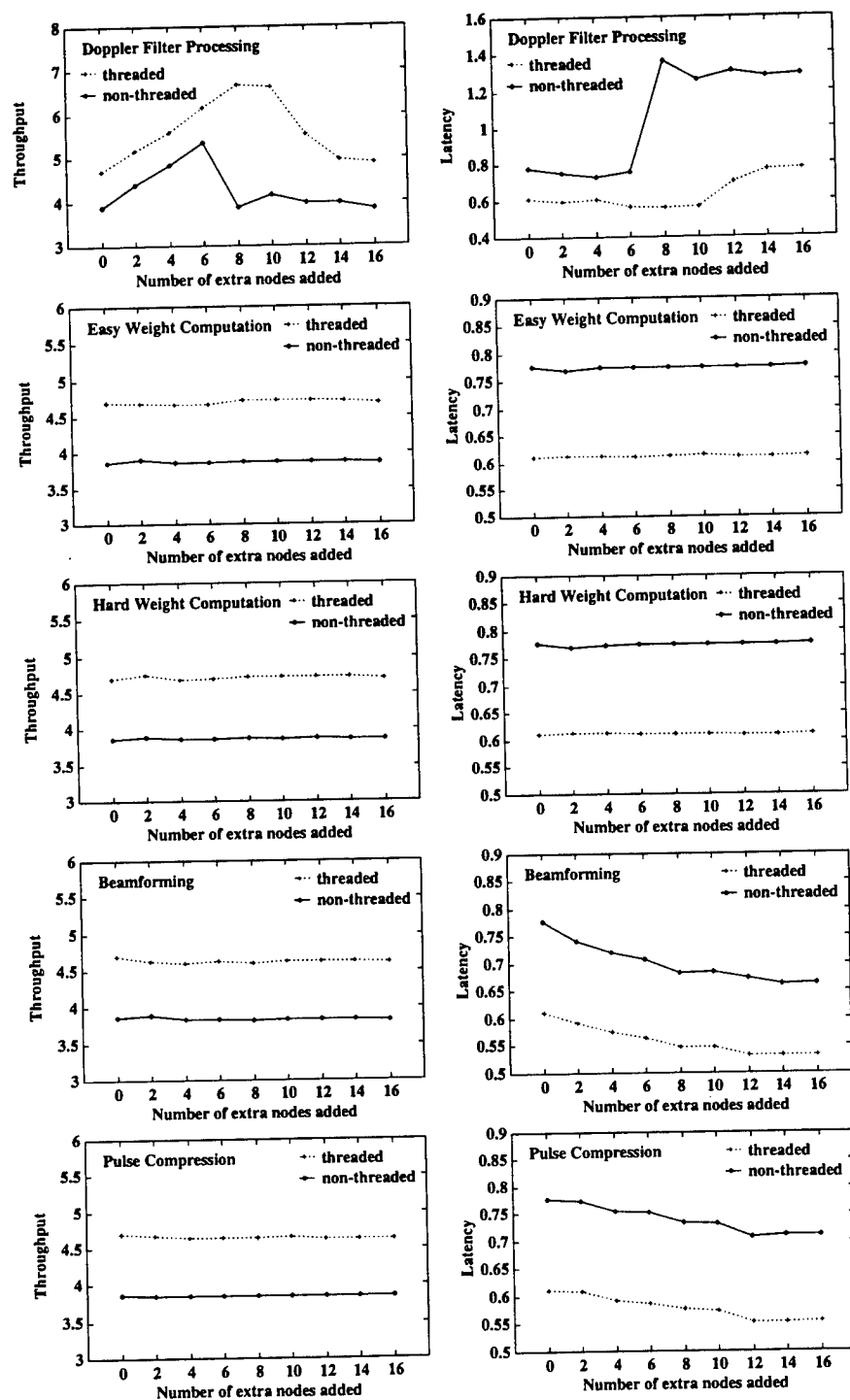


Figure 31. Throughput and latency results by adding 2 nodes at a time to each task.

Table 12. Performance results of non-threaded implementation for adding 4 more compute nodes to the Doppler processing task and 4 more compute nodes to pulse compression task to the case 2 in Table 10.

total number of nodes = 110		Time in seconds			
	# nodes	recv	comp	send	total
Doppler filter	20	.0084	.1429	.0579	.2092
easy weight	4	.0473	.1639	.0003	.2114
hard weight	56	.0708	.1404	.0003	.2115
easy BF	8	.0901	.1176	.0003	.2079
hard BF	8	.0796	.1224	.0003	.2024
pulse compr	12	.0813	.1135	.0108	.2057
CFAR	2	.0701	.1348	-	.2049
estimated	throughput		4.7271		
	latency		0.8276		
measured	throughput		4.8368		
	latency		0.6650		

nodes can be added. We observed that only the addition of nodes to the Doppler filter processing task can increase the throughput. Similarly, for the case with 102 compute nodes that has satisfied the minimum throughput requirement, only beam-forming and pulse compression tasks are candidates for the addition of more compute nodes to reduce the latency.

Compute node assignment can also be made in such a way that throughput and latency are both improved simultaneously. We again take case 2 (with 102 compute nodes) from Tables 10 and 11 as an example and add 8 more compute nodes to analyze its effect on the throughput and latency. Tables 12 and 13 show the results of adding 4 compute nodes to the Doppler filter processing task and 4 nodes to the pulse compression task. By increasing the number of compute nodes by 7.8%, the improvement in throughput is 25.1% and in latency it is 14.4% for the non-threaded implementation. Meanwhile, the threaded implementation shows 19.7% improvement in throughput and 10.6% improvement in latency. From these experimented results,

Table 13. Performance results of threaded implementation for adding 4 more compute nodes to the Doppler processing task and 4 more compute nodes to pulse compression task to the case 2 in Table 11.

total number of nodes = 110		Time in seconds			
	# nodes	recv	comp	send	total
Doppler filter	20	.0082	.1026	.0681	.1789
easy weight	4	.0905	.0998	.0004	.1907
hard weight	56	.0990	.0900	.0005	.1895
easy BF	8	.0894	.0955	.0003	.1851
hard BF	8	.0644	.1100	.0003	.1747
pulse compr	12	.0799	.0865	.0109	.1773
CFAR	2	.1023	.0736	-	.1759
estimated throughput		5.2427			
latency		0.7172			
measured throughput		5.6137			
latency		0.5458			

we can draw the following conclusions. Extra compute nodes can be assigned to the task that has the maximum execution time among all tasks. In this way, the execution time of this task is reduced and according to Equation (4), the throughput is increased. From Equation (5), latency is the sum of several tasks' execution time. Extra compute nodes can be added to those tasks which benefit the most, that is, the tasks with greatest reduced execution time when more nodes are assigned. The sum of these tasks can be reduced the most and therefore it minimizes the latency.

4.4 Summary

In this chapter we presented performance results for a PRI-staggered post-Doppler STAP algorithm implementation on the Intel Paragon machine at Air Force Research Laboratory, Rome, New York. This Paragon machine is an MP system which has

three processors on each compute node board. By taking advantage of this architecture, a multi-threaded implementation is presented and compared to the non-threaded implementation. Performance results indicate that our approach of parallel pipelined implementation scales well both in terms of throughput and latency whether the multi-threaded technique is used or not. Our design and implementation not only shows tradeoffs in parallelization, compute node assignment, and various overheads in inter-task communication etc., but it also shows that accurate performance measurement of these systems is very important.

Chapter 5

I/O Implementation

In this chapter we build upon our work in the previous chapters where we devised strategies for high performance parallel pipeline implementations, in particular, for Space-Time Adaptive Processing (STAP) applications [31, 32]. A modified Pulse Repetition Interval (PRI)-staggered post-Doppler STAP algorithm was implemented based on the parallel pipeline model and scalable performance was obtained both on the Intel Paragon and the IBM SP. Normally, this parallel pipeline system does not include disk I/O costs. Since most radar applications require signal processing in real time, thus far we have assumed that the signal data collected by radar is directly delivered to the pipeline system, as shown in the overall radar and signal processing system of Figure 32.

In practice, the I/O can be done either directly from a radar or through disk file systems. In this chapter, we focus on the I/O implementation of the parallel pipeline STAP algorithm when I/O is carried out through a disk file system. Using existing parallel file systems, we investigate the impact of I/O on the overall pipeline system performance. Two designs of I/O are employed: in the first design the I/O is embedded in the pipeline without changing the task structure and in the other a separate task is created to perform I/O operations. With different I/O strategies, we ran the parallel pipeline STAP system portably and measured the performance on the Intel Paragon at California Institute of Technology and on the IBM SP at Argonne

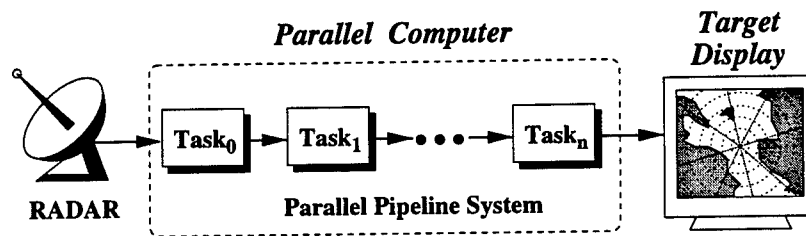


Figure 32. Data flow of a radar and signal processing system using parallel computers.

National Laboratory (ANL.) The parallel file systems on both the Intel Paragon and the IBM SP contain multiple stripe directories for applications to access disk files efficiently. On the Paragon, two PFS file systems with different stripe factors were tested and the results were analyzed to assess the effects of the size of the stripe factor on the STAP pipeline system. On the IBM SP, the performance results were obtained by using the native parallel file system, PIOFS, which has 80 stripe directories.

Comparing the two parallel file systems with different stripe sizes on the Paragon, we found that an I/O bottleneck results when a file system with smaller stripe size is used. Once a bottleneck appears in a pipeline, the throughput which is determined by the task with maximum execution time degrades significantly. On the other hand, the latency is not significantly affected by the bottleneck problem. This is because the latency depends on all the tasks in the pipeline rather than the task with the maximum execution time. Furthermore, when evaluating the performance results of the two I/O designs, we observed that the latency can be improved by merging two tasks in the pipeline. In this chapter, we also examine the possibility of improving latency by reorganizing the task structure of the STAP pipeline system.

A sequence of raw signal data sets collected by a radar form the input to the STAP pipeline system. Each of these raw data sets is in the form of a three dimensional array. However, the three dimensions of this array are not organized in a way such that each Fast Fourier Transformation (FFT) in the Doppler filter processing task can be performed in a single processor. Without special hardware support to pre-process the collected raw data, data redistribution is needed before delivering the

data to the Doppler filter processing task. In the real application we implemented, this pre-processing work includes data type conversion and corner turn on the three-dimensional array. Using a software approach, we also embedded pre-processing operation on the raw data in the two I/O designs and compared their performances.

The rest of the chapter is organized as follows: The characteristics of the parallel file systems tested are described in Section 5.1. The I/O design and implementation are presented in Section 5.2 and their performance results are given in Section 5.3. Section 5.4 presents the implementation when tasks are combined to improve latency. The software approach to pre-processes raw signal data is described in Section 5.5.

5.1 Parallel File Systems

Only input part of parallel I/O was implemented on the STAP pipeline system because most applications like STAP send their detection results to display devices in real time. The input to the STAP pipeline system is a series of CPI data sets captured by the radar. To test our parallel pipeline system with regard to I/O performance, these CPI data sets were stored in the parallel file system and provided to the pipeline system through machine's I/O nodes. We used the parallel I/O library developed by Intel Paragon and IBM SP systems to perform read operations.

5.1.1 Intel Paragon PFS File System

The Intel Paragon OSF/1 operating system provides a special file system type called PFS, for Parallel File System, which gives applications high-speed access to a large amount of disk storage [29]. PFS file systems are optimized for simultaneous access by multiple nodes. Each PFS file system consists of multiple stripe directories. Each stripe directory is the mount point of a separate UNIX file system. A PFS file system collects together several file systems into a unit that behaves like a single large file system. A file stored in PFS is distributed, or striped, across the stripe directories that make up the PFS file system. The performance of accessing a single PFS file is

significantly improved by multiple stripe devices providing disk data simultaneously. The amount of data from a PFS file that is stored in each stripe directory is determined by the PFS file system's stripe unit. The stripe units on all Paragon parallel systems at Caltech are 64K bytes. Two PFS file system were tested : one has 16 stripe directories (stripe factor 16) and the other has a stripe factor of 64.

We used the Intel Paragon NX library to implement the I/O of the parallel pipeline STAP system. Since only input part of the I/O is needed for providing a series of CPI data sets to the pipeline, only read operations are investigated. Subroutine **gopen()** was used to open CPI files globally because it offers better performance and causes less system overhead. NX library provides six I/O modes for an application to access files: **M_UNIX**, **M_LOG**, **M_SYNC**, **M_RECORD**, **M_GLOBAL**, and **M_ASYNC**. A file's I/O mode is set when the file is opened with **gopen()**. Only non-collected I/O mode **M_ASYNC** was used because it provided an efficient parallel read operation. This mode has the following characteristics on an opened PFS file:

- every node has its own file pointer
- read operations are not synchronized
- read can be for variable-length, unordered records

This mode allows multiple reads to access a single file simultaneously without agreement on record size or file offset among nodes. If read operations access exclusive portions of a file, it behaves like each compute node reads from its own file independently. In the pipeline system, the number of nodes to read CPI files may vary and, therefore, the length of the subset of CPI file for each node to read can be different. Besides, only the nodes in the first task of the pipeline system issue read operations, rather than all nodes allocated for the whole application. This explains why we used **M_ASYNC** mode and it is also the only feasible and efficient way to read disk files in parallel. All other collective I/O modes provided by the OSF/1 operating system require that all nodes in the application perform the same I/O operations and, hence, accessing files by a subset of the nodes is prohibited for these modes. In

addition, we used asynchronous I/O function calls: **iread()** and **ireadoff()** in order to overlap I/O operations with the computation and communication.

5.1.2 IBM SP PIOFS File System

The IBM AIX operating system provides a parallel file system called Parallel I/O File System (PIOFS) which is designed for IBM RS/6000 SP to allow fast parallel access to large temporary data files [33]. The PIOFS on the IBM SP at ANL is made up of 5 servers. Four of the servers have 4 Serial Storage Architecture (SSA) disks while the fifth is the directory server. Each of the 4 SSA disks is partitioned into 5 slices. Therefore, there are a total of 80 slices (striped directories) in the ANL PIOFS file system. The default basic striped unit (BSU) is 64K bytes. A file stored in the PIOFS is physically divided into several blocks with each equal to the size of one BSU, and these blocks are stored in the 80 striped directories in a round-robin manner.

IBM PIOFS supports existing C read, write, open and close functions. In addition to a UNIX-like I/O interface, PIOFS also supports logical partitioning of files. A processor can independently specify a logical view of the data in a file, a subfile, and then perform I/O on this subfile with a single call. In our STAP I/O task implementation, we store all CPI files in the ANL PIOFS using the default BSU, 64K bytes. As for the Intel Paragon, CPI files are stored across 80 striped directories in the PIOFS file system. However, unlike the Paragon NX library, asynchronous parallel read/write subroutines are not supported on IBM PIOFS. The overall performance of the STAP pipeline system will be limited by the inability to overlap I/O operations with computation and communication.

5.2 Design and Implementation

A total of four CPI data sets stored as four files in the parallel file systems were used on both the Caltech Paragon and the ANL SP. Each of the four CPI files is of size 8M bytes. On the Paragon, these files are opened globally (or collectively)

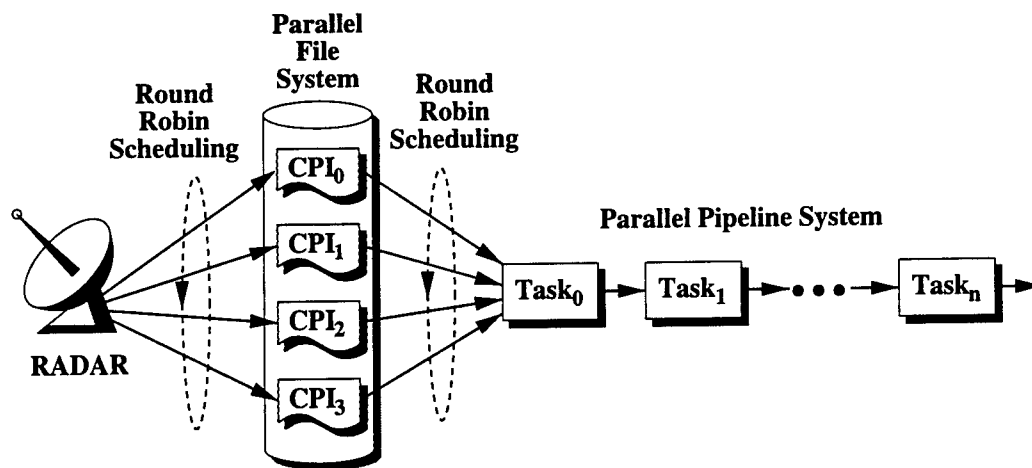


Figure 33. Four CPI data files are read from the parallel file system into the pipeline system in a round-robin manner.

by all compute nodes allocated in the whole application during the STAP pipeline system's initialization. On SP, these four files are opened only by the compute nodes that perform the I/O task. During each of the following steps after the initialization, only nodes assigned to the first task perform read operations from the parallel file system. We assume that the radar writes its collected CPI data into these four files in a round-robin manner. Similarly, the STAP pipeline system was also designed to read these four files in a round-robin fashion but at times that are different from the times at which the radar writes. This is shown in Figure 33. In this manner, the problem of data inconsistency for read/write operations between the radar and the STAP parallel pipeline system is minimized.

All nodes allocated to the first task (the I/O nodes) of the pipeline read exclusive portions of each CPI file with proper offsets. Because the number of I/O nodes may vary due to different node assignments to the I/O task, the length of data for the read operations can be different. The read length and file offset for all the read operations are set only during the STAP pipeline system's initialization and is not changed afterward. Therefore, in each of the following iterations, only one read

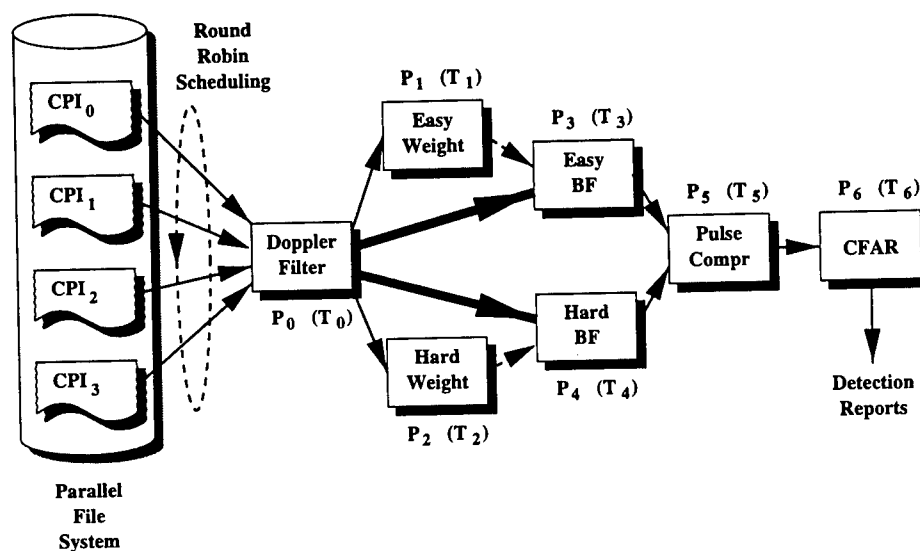


Figure 34. I/O task is embedded in the Doppler filter processing task of the STAP pipeline system.

function call is needed. On the Paragon, since asynchronous read subroutines were used, an additional subroutine waiting for the read's completion was also required in each iteration.

5.2.1 I/O Task Implementation

Two designs for the I/O task were implemented in the STAP pipeline system. The first one, shown in Figure 34, embeds the parallel I/O in the first task of the pipeline, i.e. in the Doppler filter processing task. The Doppler filter processing task now consists of three phases, reading CPI data from files, computation, and sending phases. The second I/O implementation creates a new task for reading CPI data and this task is added to the beginning of the pipeline. Figure 35 shows the structure of the overall pipeline system with this implementation. The only job of this I/O task is to read CPI data from the files and deliver it to the Doppler filter processing task.

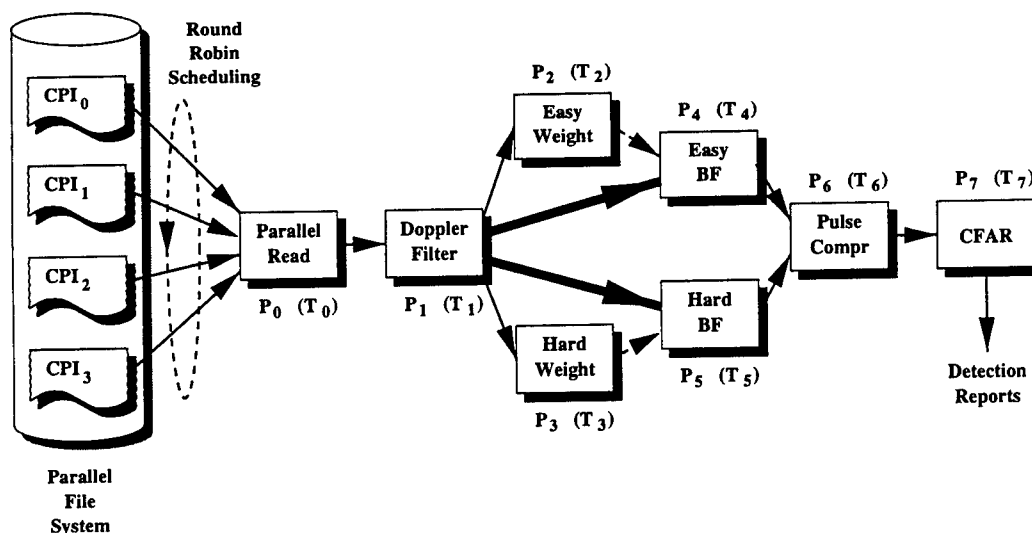


Figure 35. A separate I/O task for reading CPI data is added to the STAP pipeline system.

5.3 Performance Results

Performance results are given for the two I/O implementations on the parallel pipeline STAP system. For each implementation, parallel file systems on the Paragon and the SP were tested. On the Paragon, we used two PFS file systems, one with 16 stripe directories and the other with 64 stripe directories. On the SP, only the parallel file system with 80 striped directories was tested. On both machines, the stripe unit for the parallel file systems is 64K bytes. The size of each CPI data file is 8M bytes that results in 128 stripe units distributed across all stripe directories in all the parallel file systems.

5.3.1 I/O Embedded in the First Task

In the first I/O implementation on the Paragon, the Doppler filter processing task reads its input from CPI files using asynchronous read calls. A double buffering strategy is employed to overlap the I/O operations with computation and communication in this task. Table 14 shows the timing results for this implementation on the

Paragon PFS file system with 16 stripe directories. Three cases of node assignments to all tasks in the pipeline system are given, each doubles the number of nodes of another. The throughput scales well in the first two cases, but degrades when the total number of nodes goes up to 224. In this case, we observe that the timing results of the receive phase in the first task are relatively higher than the other two phases, the compute and send phases. The I/O operations for reading CPI data files here become a bottleneck for the pipeline system. This bottleneck forces the rest of the following tasks in the pipeline system to wait for their input data from their previous tasks.

Table 15 gives the timing results for the same cases as in Table 14, but on a Paragon PFS file system with 64 stripe directories. Both throughput and latency showed linear speedups. In the first two cases with 56 and 112 nodes, the results of throughput and latency are approximately the same for both file systems with 16 and 64 stripe directories. However, in the case with 224 nodes, we observe that the I/O bottleneck is relieved by using 64 stripe directories. The efficiency of I/O operations plays an important role in the overall performance of the pipeline system. The I/O task may become a bottleneck in the pipeline and directly affect the throughput results.

On the other hand, a linear speedup was obtained for the latency results. The I/O bottleneck problem does not affect the latency significantly. We can observe that in the case with 224 nodes, the latency of using 16 stripe directories is slightly greater than using 64 stripe directories. This can be explained by examining the throughput and latency equations, (4) and (5), shown in Section 3.2 of Chapter 3. Unlike the throughput that depends on the maximum of the execution times of all the tasks, the latency is determined by the sum of the execution times of all the tasks except for the tasks with temporal dependency. Therefore, even though the execution time of the Doppler filter processing task is increased, the delay does not contribute much to the latency. Comparing Tables 14 and 15, the latency did not degrade significantly and still scaled well in the case with 224 nodes. Figure 36 shows the performance results of this I/O design in bar charts.

Table 14. Performance results on the Paragon with the I/O embedded in the Doppler filter processing task.

PFS stripe factor = 16

case 1: total number of nodes = 56		Time in seconds			
	# nodes	recv	comp	send	total
Doppler filter	12	.0101	.2566	.0916	.3584
easy weight	3	.1317	.2214	.0002	.3534
hard weight	28	.0684	.2838	.0003	.3525
easy BF	3	.1451	.1921	.0003	.3375
hard BF	4	.1596	.1756	.0002	.3354
pulse compr	4	.1070	.1979	.0298	.3347
CFAR	2	.1983	.1361	-	.3343
throughput		2.9560			
latency		0.9804			

case 2: total number of nodes = 112		Time in seconds			
	# nodes	recv	comp	send	total
Doppler filter	24	.0178	.1292	.0663	.2134
easy weight	6	.0856	.1110	.0002	.1968
hard weight	56	.0483	.1423	.0059	.1965
easy BF	6	.0939	.0958	.0003	.1901
hard BF	8	.0906	.0885	.0003	.1795
pulse compr	8	.0648	.0993	.0150	.1792
CFAR	4	.1107	.0683	-	.1790
throughput		5.4996			
latency		0.5171			

case 3: total number of nodes = 224		Time in seconds			
	# nodes	recv	comp	send	total
Doppler filter	48	.0871	.0619	.0317	.1807
easy weight	12	.1056	.0557	.0002	.1616
hard weight	112	.0905	.0724	.0009	.1639
easy BF	12	.1080	.0482	.0003	.1565
hard BF	16	.1030	.0509	.0003	.1542
pulse compr	16	.0983	.0502	.0078	.1562
CFAR	8	.1217	.0343	-	.1561
throughput		6.2708			
latency		0.3292			

Table 15. Performance results on the Paragon with the I/O embedded in the Doppler filter processing task.

PFS stripe factor = 64

case 1: total number of nodes = 56		Time in seconds			
	# nodes	recv	comp	send	total
Doppler filter	12	.0314	.2461	.0916	.3691
easy weight	3	.1262	.2216	.0002	.3480
hard weight	28	.0628	.2840	.0003	.3471
easy BF	3	.1397	.1921	.0003	.3321
hard BF	4	.1537	.1756	.0002	.3295
pulse compr	4	.1011	.1977	.0298	.3286
CFAR	2	.1920	.1363	-	.3282
throughput	3.0111				
latency	0.9787				

case 2: total number of nodes = 112		Time in seconds			
	# nodes	recv	comp	send	total
Doppler filter	24	.0107	.1280	.0557	.1944
easy weight	6	.0787	.1111	.0020	.1917
hard weight	56	.0453	.1427	.0039	.1919
easy BF	6	.0860	.0959	.0003	.1823
hard BF	8	.0878	.0885	.0003	.1766
pulse compr	8	.0615	.0995	.0151	.1761
CFAR	4	.1077	.0682	-	.1759
throughput	5.6068				
latency	0.5143				

case 3: total number of nodes = 224		Time in seconds			
	# nodes	recv	comp	send	total
Doppler filter	48	.0069	.0673	.0309	.1052
easy weight	12	.0510	.0559	.0002	.1071
hard weight	112	.0355	.0733	.0019	.1106
easy BF	12	.0526	.0483	.0003	.1013
hard BF	16	.0471	.0515	.0003	.0989
pulse compr	16	.0407	.0503	.0080	.0990
CFAR	8	.0642	.0343	-	.0985
throughput	10.0262				
latency	0.2871				

Table 16. Performance results on the SP with the I/O embedded in the Doppler filter processing task.

PIOFS stripe factor = 80

case 1: total number of nodes = 18					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	6	.1172	.0734	.1966	.3872
easy weight	1	.2717	.1070	.0001	.3788
hard weight	7	.1590	.2194	.0002	.3786
easy BF	1	.2927	.0829	.0001	.3757
hard BF	1	.2595	.1177	.0002	.3775
pulse compr	1	.2230	.1545	.0001	.3776
CFAR	1	.2941	.0828	-	.3770
throughput	2.6715				
latency	1.2353				

case 2: total number of nodes = 30					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	8	.1109	.0543	.1031	.2683
easy weight	1	.1471	.1045	.0002	.2518
hard weight	14	.1523	.1072	.0002	.2597
easy BF	2	.2189	.0412	.0001	.2602
hard BF	2	.1999	.0606	.0001	.2606
pulse compr	2	.1801	.0777	.0001	.2579
CFAR	1	.1801	.0801	-	.2602
throughput	3.8319				
latency	0.7810				

case 3: total number of nodes = 60					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	16	.1044	.0304	.0474	.1823
easy weight	2	.1314	.0547	.0001	.1862
hard weight	28	.1303	.0566	.0002	.1871
easy BF	4	.1571	.0219	.0002	.1792
hard BF	4	.1492	.0298	.0002	.1792
pulse compr	4	.1370	.0396	.0001	.1767
CFAR	2	.1399	.0403	-	.1802
throughput	5.5364				
latency	0.5004				

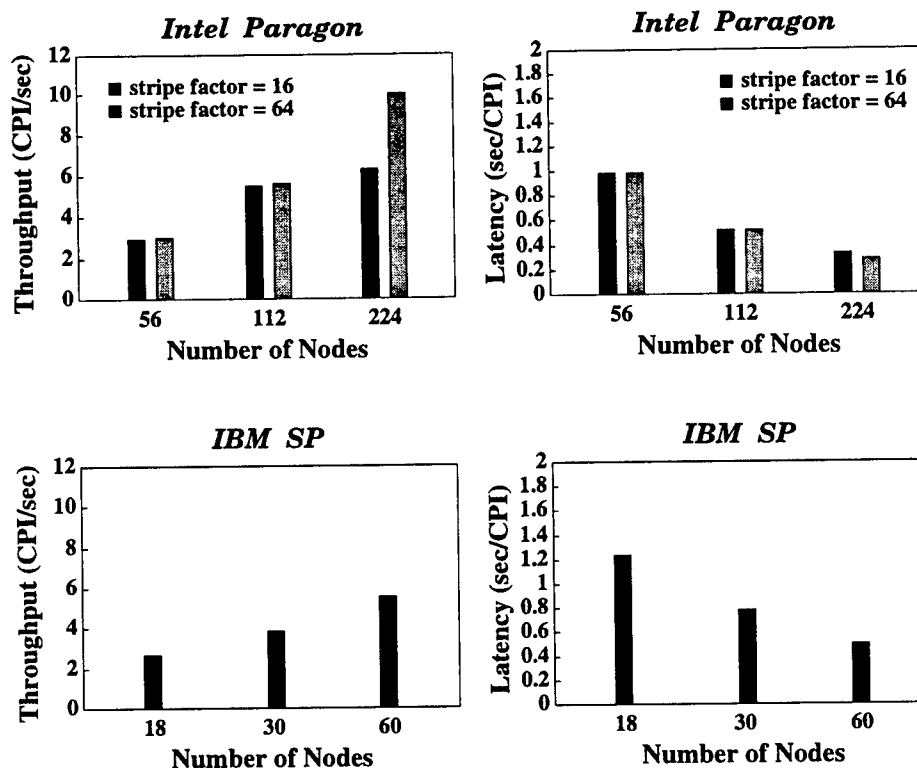


Figure 36. Performance results for the STAP pipeline system with parallel I/O embedded in the Doppler filter processing task. This figure corresponds to Tables 14, 15, and 16.

Detailed timing results for the IBM SP at ANL are given in Table 16. The stripe factor of the PIOFS file system is 80. Because PIOFS does not provide asynchronous read/write subroutines, the I/O operations do not overlap with computation and communication in the Doppler filter processing task. Hence, the performance results for throughput and latency on the SP did not show the scalability as on the Paragon, even though the SP has faster CPUs.

5.3.2 A New I/O Task

In the second I/O task implementation, a new task is added to the beginning of the pipeline. This new task only performs the operations of reading CPI files and

distributing CPI data to its successor task, Doppler filter processing task. The STAP pipeline system then has a total of 8 tasks. Tables 17, 18, and 19 show the performance results for this I/O design. Corresponding to Tables 14, 15, and 16, all tasks have the same numbers of nodes assigned, except for the I/O task. The I/O bottleneck problem still occurs when using the Paragon PFS system with 16 stripe directories. When using the file system with 64 stripe directories, the throughput results improved. The bar charts shown in Figure 37 represent the throughput and latency results of Tables 17, 18, and 19.

Comparing the two I/O designs, we observe that the throughput results are approximately the same for both implementations. However, the latency results for the separate I/O task design are worse than the embedded implementation. This phenomenon can be explained by examining the throughput and latency equations. The equations for the throughput and latency for the STAP pipeline system are

$$throughput_8 = \frac{1}{\max_{0 \leq i < 8} T_i} \quad (8)$$

and

$$latency_8 = T_0 + T_1 + \max(T_4, T_5) + T_6 + T_7, \quad (9)$$

where T_i is the execution time for the task i .

The throughput of a pipeline system is determined by the task with the maximum execution time among all the tasks. From Tables 17 and 18, we observe that the Doppler filter processing task has the maximum execution time among all the tasks in the cases with a total of 60 and 120 nodes. In the case of 240 nodes on the PFS file system with 16 stripe directories, the maximum execution time occurs in the parallel I/O task. Using PFS with 64 stripe directories, the hard weight computation task has the maximum execution time in the case of 240 nodes. Compared to Tables 14 and 15, the throughput results have no significant change because the tasks with the maximum execution time are the same for every corresponding pair in all cases. All these tasks have the same number of compute nodes assigned and hence have approximately the same computation time. Therefore, the execution times of these

Table 17. Performance results on the Paragon with the I/O implemented as a separate task.

PFS stripe factor = 16

case 1: total number of nodes = 60				Time in seconds	
	# nodes	recv	comp	send	total
Parallel read	4	.0191	-	.3997	.4187
Doppler filter	12	.0122	.3240	.2375	.5738
easy weight	3	.2032	.2217	.0002	.4252
hard weight	28	.1390	.2846	.0003	.4239
easy BF	3	.2210	.1911	.0003	.4124
hard BF	4	.2327	.1753	.0003	.4083
pulse compr	4	.1800	.1977	.0295	.4072
CFAR	2	.2706	.1362	-	.4068
throughput				2.4127	
latency				1.9186	

case 2: total number of nodes = 120				Time in seconds	
	# nodes	recv	comp	send	total
Parallel read	8	.0559	-	.1604	.2163
Doppler filter	24	.0254	.1221	.0839	.2313
easy weight	6	.0920	.1110	.0004	.2034
hard weight	56	.0526	.1432	.0045	.2003
easy BF	6	.1003	.0960	.0003	.1966
hard BF	8	.0918	.0928	.0003	.1849
pulse compr	8	.0727	.0999	.0151	.1877
CFAR	4	.1185	.0683	-	.1867
throughput				5.3883	
latency				0.9226	

case 3: total number of nodes = 240				Time in seconds	
	# nodes	recv	comp	send	total
Parallel read	16	.1269	-	.0276	.1545
Doppler filter	48	.0833	.0463	.0245	.1541
easy weight	12	.0891	.0558	.0002	.1451
hard weight	112	.0749	.0724	.0004	.1477
easy BF	12	.0975	.0485	.0003	.1463
hard BF	16	.0924	.0516	.0003	.1443
pulse compr	16	.0869	.0502	.0077	.1448
CFAR	8	.1104	.0343	-	.1447
throughput				6.8438	
latency				0.3890	

Table 18. Performance results on the Paragon with the I/O implemented as a separate task.

PFS stripe factor = 64

case 1: total number of nodes = 60				Time in seconds	
	# nodes	recv	comp	send	total
Parallel read	4	.0628	-	.3391	.4019
Doppler filter	12	.0085	.2670	.1755	.4510
easy weight	3	.1425	.2217	.0002	.3645
hard weight	28	.0763	.2847	.0003	.3613
easy BF	3	.1621	.1914	.0003	.3537
hard BF	4	.1740	.1759	.0002	.3501
pulse compr	4	.1213	.1980	.0296	.3489
CFAR	2	.2125	.1362	-	.3488
throughput	2.8234				
latency	1.7309				

case 2: total number of nodes = 120				Time in seconds	
	# nodes	recv	comp	send	total
Parallel read	8	.0362	-	.1685	.2047
Doppler filter	24	.0280	.1084	.0786	.2151
easy weight	6	.0816	.1111	.0024	.1951
hard weight	56	.0461	.1438	.0003	.1903
easy BF	6	.0914	.0959	.0003	.1877
hard BF	8	.0891	.0908	.0003	.1802
pulse compr	8	.0672	.0999	.0151	.1822
CFAR	4	.1131	.0683	-	.1815
throughput	5.5262				
latency	0.9137				

case 3: total number of nodes = 240				Time in seconds	
	# nodes	recv	comp	send	total
Parallel read	16	.0171	-	.0617	.0788
Doppler filter	48	.0073	.0502	.0290	.0864
easy weight	12	.0503	.0558	.0002	.1063
hard weight	112	.0305	.0724	.0029	.1057
easy BF	12	.0491	.0489	.0004	.0984
hard BF	16	.0417	.0540	.0004	.0961
pulse compr	16	.0393	.0502	.0078	.0973
CFAR	8	.0629	.0343	-	.0972
throughput	10.2111				
latency	0.5193				

Table 19. Performance results on the SP with the I/O implemented as a separate task.

PIOFS stripe factor = 80

case 1: total number of nodes = 20		Time in seconds			
	# nodes	recv	comp	send	total
Parallel read	2	.1787	-	.1413	.3200
Doppler filter	6	.0045	.0724	.2548	.3316
easy weight	1	.2269	.1047	.0001	.3317
hard weight	7	.1165	.2150	.0013	.3329
easy BF	1	.0641	.0822	.2082	.3545
hard BF	1	.0416	.1179	.1874	.3469
pulse compr	1	.1459	.1538	.0656	.3653
CFAR	1	.2926	.0801	-	.3727
throughput		2.6670			
latency		2.6715			

case 2: total number of nodes = 34		Time in seconds			
	# nodes	recv	comp	send	total
Parallel read	4	.1230	-	.0594	.1823
Doppler filter	8	.0264	.0549	.0913	.1726
easy weight	1	.0639	.1043	.0001	.1683
hard weight	14	.0598	.1090	.0003	.1692
easy BF	2	.0576	.0415	.0814	.1805
hard BF	2	.0593	.0596	.0579	.1768
pulse compr	2	.0278	.0784	.0803	.1864
CFAR	1	.1092	.0804	-	.1896
throughput		5.2819			
latency		1.2766			

case 3: total number of nodes = 68		Time in seconds			
	# nodes	recv	comp	send	total
Parallel read	8	.1100	-	.0185	.1285
Doppler filter	16	.0455	.0283	.0631	.1369
easy weight	2	.0901	.0535	.0001	.1437
hard weight	28	.0839	.0554	.0001	.1395
easy BF	4	.1158	.0208	.0035	.1401
hard BF	4	.0813	.0483	.0089	.1385
pulse compr	4	.1008	.0391	.0054	.1453
CFAR	2	.1074	.0404	-	.1478
throughput		6.5063			
latency		0.6531			

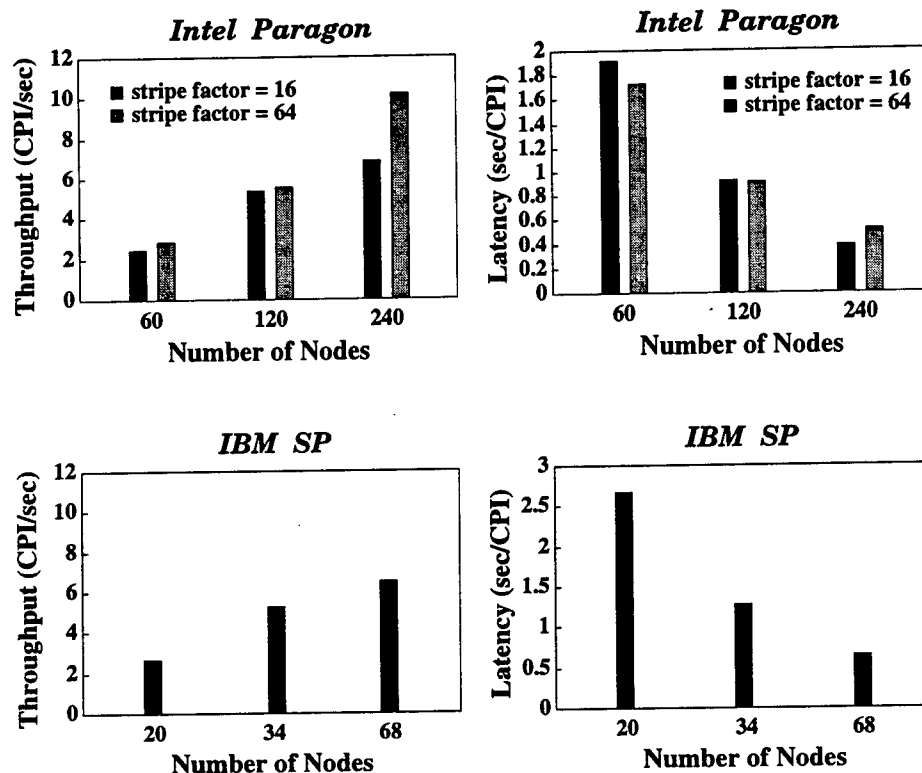


Figure 37. Performance results for the implementation using a separate I/O task. This figure corresponds to Tables 17, 18, and 19.

tasks have no significant differences for both cases and the throughput results do not change significantly.

The latency, on the other hand, is the sum of the execution times of all the tasks except for the tasks with temporal data dependency, that is, easy and hard weight computation tasks (T_2 and T_3 , respectively.) In the design with a separate I/O task, the latency contains one more term than the embedded I/O implementation: the execution time of the new task, T_0 . Therefore, the latency results become worse in this implementation.

5.4 Task Combination

From the comparison of performance results for the two I/O task implementations, we notice that the structure of the STAP pipeline system can be reorganized to improve the latency. The first implementation that embeds I/O in the Doppler filter processing task can be viewed as combining the first two tasks of the second implementation that uses a separate task for I/O. As shown in Section 5.3.2, the first I/O implementation has a better latency performance, while the throughput results are approximately the same.

5.4.1 Improving Latency

We investigate whether the latency can be further improved by combining multiple tasks of the pipeline into a single task. We consider Tables 14, 15, and 16 as an example and combine the last two tasks, the pulse compression and CFAR processing tasks, into a single task. In order to make a fair comparison, we keep the total number of nodes allocated to the whole pipeline system to be the same. The number of nodes assigned to this single task is equal to the sum of the nodes assigned to the two tasks in the original pipeline. In this case, no communication costs between pulse compression and CFAR processing tasks are incurred. Tables 20, 21, and 22 give the timing results corresponding to Tables 14, 15, and 16 with the same total number of nodes assigned to the pipeline system. Figure 38 shows the bar charts of the throughput and latency results for Tables 20, 21, and 22. Figure 39 gives a comparison of performance results of the STAP pipeline system with and without task combining. We observe that the latency improves for all cases on both Paragon PFS and SP PIOFS file systems when the last two tasks are combined.

This improvement can also be explained by examining the latency equation. Before task combination, the latency equation for the STAP pipeline system with 7 tasks is

$$latency_7 = T_0 + \max(T_3, T_4) + T_5 + T_6. \quad (10)$$

Table 20. Performance results on the Paragon with pulse compression and CFAR tasks combined.

PFS stripe factor = 16

case 1: total number of nodes = 56					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	12	.0094	.2589	.0908	.3591
easy weight	3	.1307	.2230	.0002	.3540
hard weight	28	.0660	.2868	.0003	.3531
easy BF	3	.1449	.1930	.0003	.3382
hard BF	4	.1616	.1756	.0003	.3375
PC + CFAR	6	.1517	.1863	-	.3380
throughput	2.9243				
latency	0.7913				

case 2: total number of nodes = 112					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	24	.0194	.1294	.0656	.2145
easy weight	6	.0831	.1111	.0002	.1944
hard weight	56	.0468	.1427	.0046	.1940
easy BF	6	.0914	.0958	.0003	.1874
hard BF	8	.0892	.0887	.0004	.1784
PC + CFAR	12	.0869	.0935	-	.1804
throughput	5.5340				
latency	0.4221				

case 3: total number of nodes = 224					Time in seconds
	# nodes	recv	comp	send	total
Doppler filter	48	.0953	.0623	.0323	.1900
easy weight	12	.1056	.0558	.0003	.1617
hard weight	112	.0930	.0726	.0004	.1661
easy BF	12	.1116	.0484	.0003	.1603
hard BF	16	.1063	.0513	.0004	.1579
PC + CFAR	24	.1079	.0513	-	.1592
throughput	6.1478				
latency	0.2948				

Table 21. Performance results on the Paragon with pulse compression and CFAR tasks combined.

PFS stripe factor = 64

case 1: total number of nodes = 56 Time in seconds

	# nodes	recv	comp	send	total
Doppler filter	12	.0319	.2485	.0915	.3718
easy weight	3	.1265	.2218	.0002	.3485
hard weight	28	.0631	.2839	.0003	.3473
easy BF	3	.1400	.1921	.0003	.3324
hard BF	4	.1533	.1756	.0003	.3292
PC + CFAR	6	.1449	.1860	-	.3309
throughput	3.0027				
latency	0.7957				

case 2: total number of nodes = 112 Time in seconds

	# nodes	recv	comp	send	total
Doppler filter	24	.0104	.1301	.0528	.1933
easy weight	6	.0774	.1111	.0002	.1887
hard weight	56	.0438	.1427	.0022	.1886
easy BF	6	.0853	.0959	.0003	.1815
hard BF	8	.0869	.0886	.0004	.1759
PC + CFAR	12	.0838	.0936	-	.1773
throughput	5.6029				
latency	0.4197				

case 3: total number of nodes = 224 Time in seconds

	# nodes	recv	comp	send	total
Doppler filter	48	.0071	.0676	.0306	.1054
easy weight	12	.0522	.0559	.0002	.1083
hard weight	112	.0347	.0730	.0031	.1108
easy BF	12	.0533	.0482	.0004	.1018
hard BF	16	.0481	.0512	.0003	.0997
PC + CFAR	24	.0489	.0514	-	.1003
throughput	9.8853				
latency	0.2392				

Table 22. Performance results on the SP with pulse compression and CFAR tasks combined.

PIOFS stripe factor = 80

case 1: total number of nodes = 18 Time in seconds

	# nodes	recv	comp	send	total
Doppler filter	6	.1320	.0728	.1894	.3942
easy weight	1	.2844	.1023	.0001	.3868
hard weight	7	.1738	.2131	.0002	.3870
easy BF	1	.3039	.0823	.0001	.3862
hard BF	1	.2677	.1182	.0002	.3862
PC + CFAR	2	.2683	.1194	-	.3877
throughput	2.5754				
latency	0.9388				

case 2: total number of nodes = 30 Time in seconds

	# nodes	recv	comp	send	total
Doppler filter	8	.1105	.0550	.1055	.2710
easy weight	1	.1711	.1026	.0002	.2739
hard weight	14	.1570	.1077	.0002	.2649
easy BF	2	.2225	.0417	.0001	.2644
hard BF	2	.2051	.0608	.0002	.2661
PC + CFAR	3	.1878	.0793	-	.2671
throughput	3.7492				
latency	0.6255				

case 3: total number of nodes = 60 Time in seconds

	# nodes	recv	comp	send	total
Doppler filter	16	.1044	.0279	.0462	.1786
easy weight	2	.1350	.0515	.0002	.1867
hard weight	28	.1238	.0568	.0002	.1808
easy BF	4	.1582	.0210	.0002	.1794
hard BF	4	.1485	.0300	.0003	.1787
PC + CFAR	6	.1397	.0414	-	.1810
throughput	5.5356				
latency	0.4207				

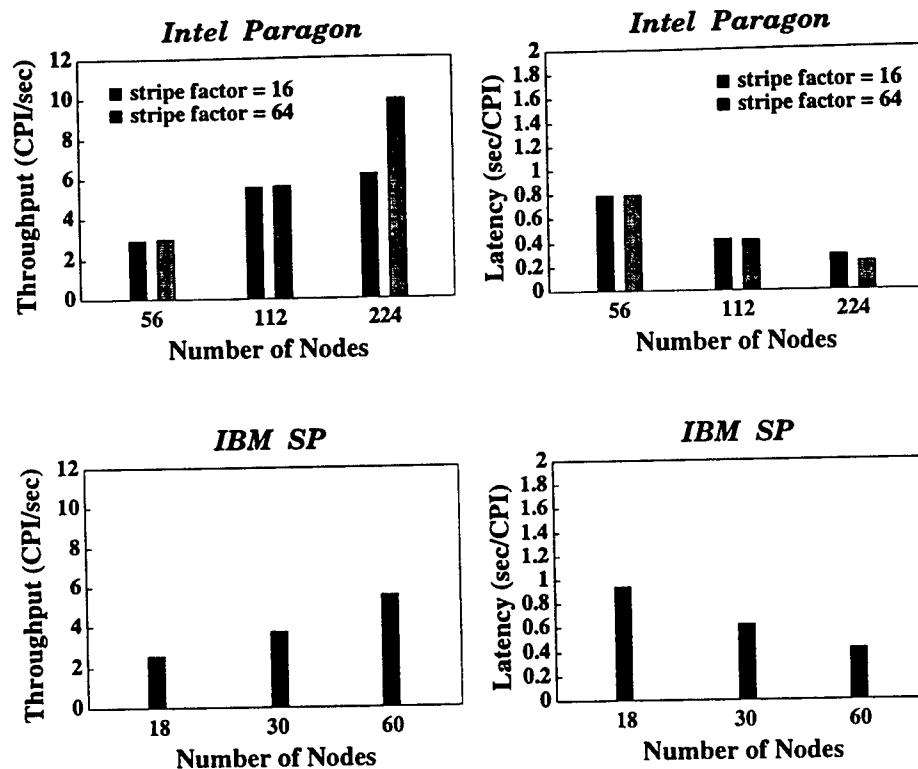


Figure 38. Performance results for the STAP pipeline system that combines the pulse compression and CFAR tasks into a single task. This figure corresponds to Tables 20, 21, and 22.

Let W_5 and W_6 be the workloads for tasks 5 and 6, respectively. The execution times for task 5 and 6 are

$$T_5 = \frac{W_5}{P_5} + C_5 + V_5 \quad (11)$$

and

$$T_6 = \frac{W_6}{P_6} + C_6 + V_6 \quad (12)$$

where C_i and V_i represent the communication time and the other parallelization overhead for task i respectively. Similarly, let T_{5+6} be the execution time of the task

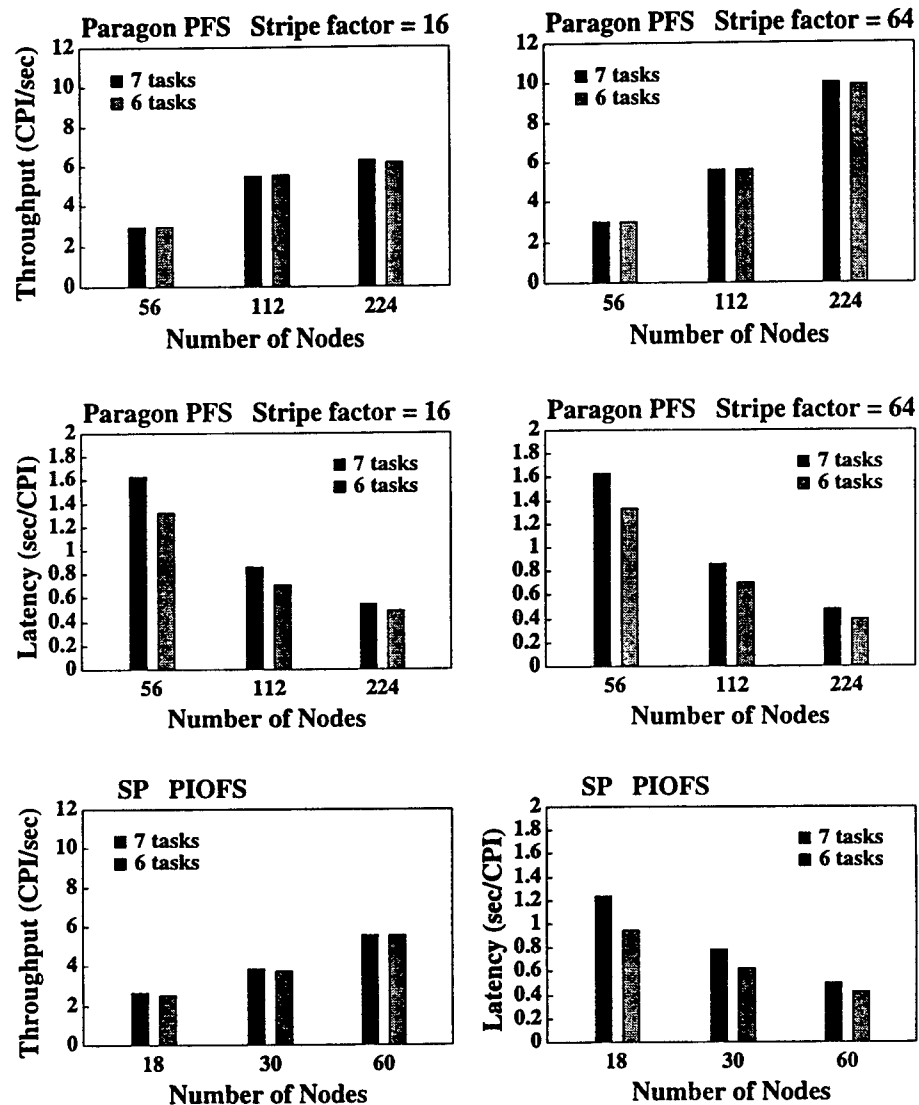


Figure 39. Performance comparison of the pipeline system with and without task combining. The throughput results remain approximately the same. Latency is improved when the last two tasks are combined.

that combines tasks 5 and 6 running on $P_5 + P_6$ nodes:

$$T_{5+6} = \frac{W_5 + W_6}{P_5 + P_6} + C_{5+6} + V_{5+6}. \quad (13)$$

By subtracting Equations (11) and (12) from Equation (13), we have

$$\begin{aligned} & T_{5+6} - (T_5 + T_6) \\ &= \frac{W_5 + W_6}{P_5 + P_6} - \frac{W_5}{P_5} - \frac{W_6}{P_6} \\ &+ C_{5+6} - C_5 - C_6 \\ &+ V_{5+6} - V_5 - V_6 \end{aligned} \quad (14)$$

where

$$\begin{aligned} & \frac{W_5 + W_6}{P_5 + P_6} - \frac{W_5}{P_5} - \frac{W_6}{P_6} \\ &= \frac{-W_5 P_6^2 - W_6 P_5^2}{P_5 P_6 (P_5 + P_6)} \\ &< 0. \end{aligned} \quad (15)$$

Communication for the combined task occurs only when receiving data from tasks 3 and 4. Prior to the task combination, the same communication takes place in the receive phase of task 5. The difference is the number of nodes used between the two tasks. Since $P_{5+6} > P_5$, the data size for each received message from tasks 3 and 4 to the combined task is smaller than that for task 5. Besides, in task 5, C_5 includes the communication cost of sending messages from task 5 to task 6 which does not occur in the combined task. Hence, we have

$$C_{5+6} < C_5. \quad (16)$$

The remaining overhead, V_i , is due to parallelization of task i . Since the operations in tasks 5 and 6 are sets of individual subroutines which require no communication within each single task, parallelization is carried out by evenly partitioning these subroutines among the nodes assigned. Due to this computational structure, the

Table 23. Percentage of latency improvement when the Pulse compression and CFAR tasks are combined into a single task.

Paragon: PFS

# nodes	56	112	224
16 stripe dir	19.3%	18.4%	10.4%
64 stripe dir	18.7%	18.4%	16.7%

SP: PIOFS

# nodes	18	30	60
80 stripe dir	24.0%	19.9%	15.9%

overhead for these two tasks becomes negligible compared to their communication costs. From Equations (14), (15), and (16) we can conclude that

$$T_{5+6} < T_5 + T_6. \quad (17)$$

Therefore, the new latency equation of the STAP pipeline system with the last two tasks combined becomes

$$\begin{aligned} \text{latency}_6 &= T_0 + \max(T_3, T_4) + T_{5+6} \\ &< \text{latency}_7 \end{aligned} \quad (18)$$

Combining the last two tasks, therefore, reduces the latency.

Table 23 gives the percentage of improvement in latency when the last two tasks are combined. These improvements were made without adding any extra nodes to the pipeline system. We observe that the percentage decreases as the number of nodes goes up. Normally, scalability of the parallelization tends to decrease when more processors are used. This also explains the trend for the percentage improvement shown in Table 23. Notice that the tasks that can be combined to improve the latency do not include tasks with temporal data dependency. It is because only those tasks with spatial data dependency contribute to the latency.

5.4.2 Improving Throughput

The throughput results, on the other hand, do not change significantly when the two tasks are combined. This is because the throughput is determined by the task with the maximum execution time among all the tasks, which is still the maximum in the new pipeline system. Assuming that T_{max} is the maximum execution time before task combination, the throughput is given by

$$throughput_7 = \frac{1}{T_{max}}$$

where

$$\begin{aligned} T_{max} &= \max_{0 \leq i < 7} T_i \\ &\geq \max(T_5, T_6) \end{aligned}$$

From Equations (11), (12), and (13), the execution time of the new combined task becomes

$$\begin{aligned} T_{5+6} &\approx \frac{P_5 T_5 + P_6 T_6}{P_5 + P_6} \\ &\leq \frac{P_5 \max(T_5, T_6) + P_6 \max(T_5, T_6)}{P_5 + P_6} \\ &= \max(T_5, T_6) \end{aligned} \tag{19}$$

and the new maximum execution time becomes

$$\begin{aligned} T'_{max} &= \max(T_0, T_1, T_2, T_3, T_4, T_{5+6}) \\ &\leq \max(T_0, T_1, T_2, T_3, T_4, T_5, T_6) \\ &= T_{max}. \end{aligned}$$

Therefore, the throughput will not decrease after task combination because

$$\begin{aligned} throughput_6 &= \frac{1}{T'_{max}} \\ &\geq \frac{1}{T_{max}} \\ &= throughput_7. \end{aligned} \tag{20}$$

Both latency and throughput can be improved simultaneously when one of the combined tasks determines the throughput of the pipeline system. Suppose that either task 5 or task 6 has the maximum execution time among all the 7 tasks in the STAP pipeline system, that is,

$$\begin{aligned} T_{max} &= \max(T_5, T_6) \\ &> \max_{0 \leq i \leq 4} T_i. \end{aligned} \quad (21)$$

Notice that none of these two tasks has temporal data dependency. From Equation (18), we have latency improvement when tasks 5 and 6 are combined. From Equations (20) and (21), the throughput is increased. The reduction of execution time of both tasks 5 and 6 contributes to the latency as well as to the throughput. Therefore, not only the throughput can be increased, but the latency can be also reduced. Note that in our experiment results shown in the previous section, the task with the maximum execution time is neither task 5 nor task 6, that is, $T_{max} > \max(T_5, T_6)$.

5.5 Raw CPI Data Redistribution

The presentation in this chapter up to now assumed that a special hardware is available to pre-process the raw CPI data received by the radar before delivering it to the STAP pipeline system. However, this special purpose equipment may not perform very efficiently or may not be available. We investigate the possibility of implementing this data pre-processing operation using a software approach. Actually, Air Force Research Laboratory (AFRL) performed a real time STAP demonstration using exactly the same signal processing algorithm as ours onboard an airborne platform in May 1996 [12, 13]. The radar was a phased array L-Band radar with 32 elements organized into two rows of 16 each. Only the data from the upper 16 elements were processed with STAP. This data is a 1.25 MHz intermediate frequency (IF) signal that is 4:1 oversampled at 5 MHz. The number representation at IF is 14 bits, 2's complement and is converted to 16 bit baseband real and imaginary numbers. Special interface boards were used to digitally demodulate to baseband. The signal data

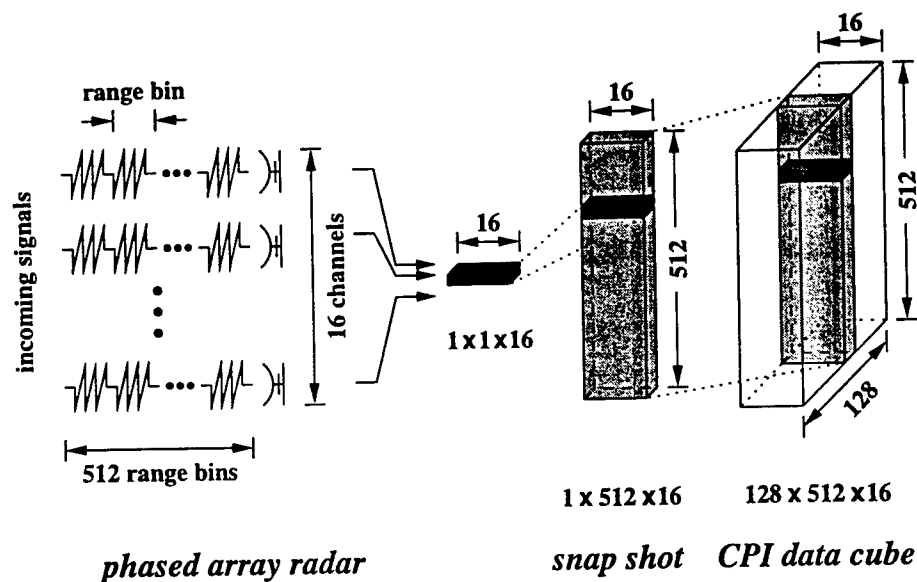


Figure 40. Raw CPI data received from a phased array radar is used to form a $128 \times 512 \times 16$ three dimensional data cube.

formed a raw 3-dimensional data cube called coherent processing interval (CPI) data cube comprised of 128 pulses, 512 range gates (32.8 miles), and 16 channels, shown in Figure 40. These special interface boards were also used to corner turn the data cube so that CPI is unit stride along pulses. It speeds the subsequent Doppler processing on the High Performance Computing (HPC) systems. Live CPI data from a phased-array radar were processed by a ruggedized version of the Paragon computer. The STAP algorithm was performed on this computer using the raw data from the 16 columns of the phased array.

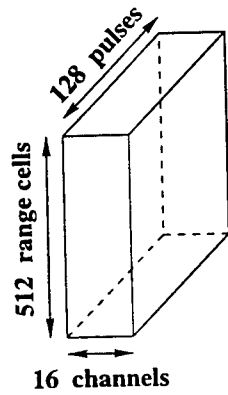
All experiments described in the previous sections assumed that this special purpose hardware was used to pre-process the raw CPI data such that each CPI data cube is corner-turned from $128 \times 512 \times 16$ to $512 \times 16 \times 128$ and each complex element in a CPI is type-converted from two 16-bits real numbers to two 32-bits real numbers (type float in C language.) The operations of corner turn and CPI data partitioning among compute nodes are illustrated in Figure 41. The reason for the corner turn

operations is that the major operations in the Doppler filter processing task, the Fast Fourier transforms (FFTs), need to be performed along the pulse dimension of the CPI cube. That is, 128-point FFTs are performed for every range and channel. The corner turn operation, here, is to allow each FFT to be computed on a single compute node in the Doppler filter processing task. Given this hardware, the parallel pipeline STAP system can directly process the CPI data without redistributing it among the compute nodes once the CPI data is read from the disk.

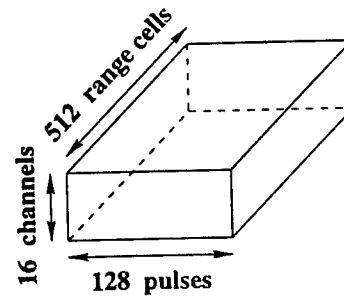
5.5.1 Corner Turn and Type Conversion

Without hardware support for the operations of corner turn and type conversion, the parallel pipeline STAP system has to include this in its implementation. In order that every FFT can be processed in a single compute node in the Doppler filter processing task, the CPI data has to be partitioned along the dimension of range cells among the compute nodes assigned, shown in Figure 41(d). Note that two consecutive pulses in a raw CPI data cube are stored in disks at a distance of $512 \cdot 16$ complex numbers. By partitioning the raw CPI along the range dimension, each sub-CPI data for one node consists of several pieces of non-contiguous data. For instance, we use 4 nodes to read a raw CPI data cube and it results in a sub-CPI of size $128 \times 128 \times 16$. That is, each sub-CPI has 128 pieces of data and each piece is of size 128×16 . Although contents of each data piece are stored contiguously in disks, the 128 data pieces themselves are not adjacent to each other. To obtain the sub-CPI data required by each node, two implementations for reading CPI data can be done:

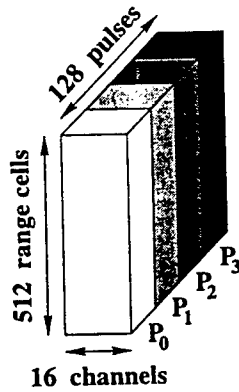
1. Every node performs several read operations directly from the disks. Each read is for a data piece of a sub-CPI. After the sub-CPI data is read, type-conversion operations are applied.
2. Using a two-phase I/O access strategy [34], the CPI data is first read using data distribution which conforms with the distribution of CPI data over the disks. This results in each node making a single, large, and contiguous disk space



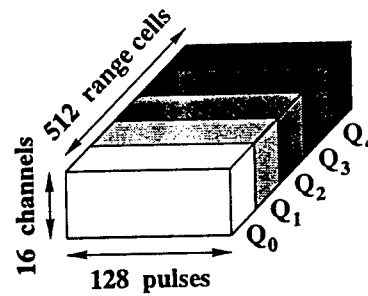
(a) Raw CPI : $128 \times 512 \times 16$



(b) Corner-turned CPI : $512 \times 16 \times 128$



(c) CPI partitioned in I/O task



(d) CPI partitioned in Doppler filter processing task

Figure 41. (a) Raw CPI data received from the radar as a $128 \times 512 \times 16$ data cube. (b) Corned-turned CPI data cube of size $512 \times 16 \times 128$. (c) Raw CPI partitioned among 4 reading nodes. (d) Corned-turned CPI partitioned among 5 nodes.

access. In the second phase, the sub-CPI data is type-converted, corner-turned, and redistributed among the nodes to match the desired data distribution.

Two-phase I/O access strategy has been shown to improve the I/O performance significantly. This method first reduces the I/O bottleneck from disks to compute nodes by making all the file accesses large and contiguous. Second, the data redistribution uses the inter-processor communication network with higher bandwidth and higher degree of connectivity.

5.5.2 Implementation

To read CPI files in parallel, we implemented the two-phase I/O access strategy on the two STAP pipeline system I/O designs described in Section 5.2. The implementation for the reading of CPI files for the STAP pipeline system with a separate I/O task is shown in Figure 42. In this implementation, each node in the I/O task performs the following steps:

1. uses one read operation to read an exclusive part of CPI data. In other words, the CPI data is partitioned into exclusive subsets and node i in the I/O task reads the i^{th} subset of each CPI file.
2. performs the corner turn and type conversion operations on the sub-CPI data.
3. redistributes the sub-CPI data with other nodes in the I/O task such that each node receives all parts of sub-CPI data it is responsible for. Data exchange in this step is an all-to-all personalized communication within the same group of nodes.
4. sends the re-organized sub-CPI data to the Doppler filter processing task. The communication pattern in this step is a left-right shift communication. Notice that the number of nodes assigned to the I/O task may be different from the Doppler filter processing task.

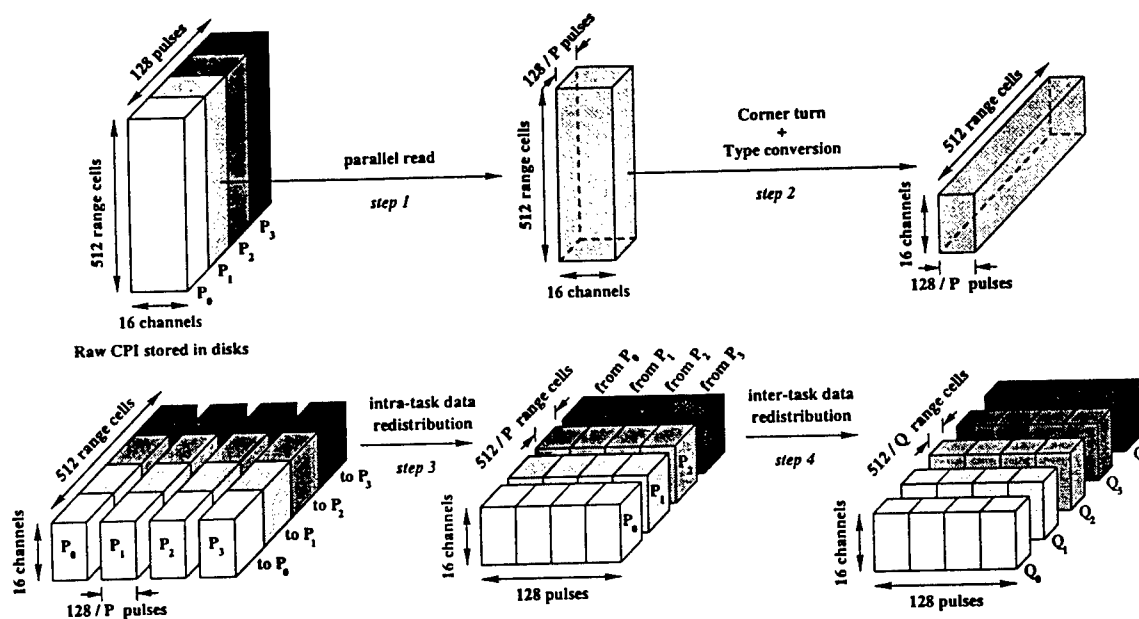


Figure 42. Implementation of parallel reading of raw CPI data from disks and its distribution for the Doppler filter processing task.

In the first I/O design that embeds the I/O in the Doppler filter processing task, the only difference is that it is without step 4, the left-right shift communication. In addition, all the steps are performed within the same group of nodes. The sub-CPI data redistribution is performed within the same group of compute nodes in the Doppler filter processing task. As opposed to the inter-task data dependency discussed in Section 2.4.1 of Chapter 2, this data redistribution results in an intra-task data dependency. The intra-task dependency exists when intermediate results need to be exchanged during the execution of a single parallel task in the pipeline.

5.5.3 Performance Results

The performance results for the implementation using a separate I/O task are given in Tables 24 and 25, for Paragon PFS file systems with 16 and 64 striped directories, respectively. Figure 43 shows the bar charts corresponding to Tables 24 and 25.

Table 24. Performance results on the Paragon with the I/O implemented as a separate task in which the corner turn and type conversion are embedded in the receive phase.
PFS stripe factor = 16.

case 1: total number of nodes = 64					Time in seconds
	# nodes	recv	comp	send	total
Parallel read	8	.3256	-	.0003	.3259
Doppler filter	12	.0634	.1744	.0907	.3285
easy weight	3	.1053	.2215	.0002	.3270
hard weight	28	.0403	.2849	.0003	.3255
easy BF	3	.1204	.1923	.0003	.3131
hard BF	4	.1346	.1757	.0003	.3105
pulse compr	4	.0812	.1978	.0302	.3092
CFAR	2	.1726	.1361	-	.3087
throughput	3.2079				
latency	1.2516				

case 2: total number of nodes = 128					Time in seconds
	# nodes	recv	comp	send	total
Parallel read	16	.1485	-	.0099	.1585
Doppler filter	24	.0037	.0976	.0580	.1593
easy weight	6	.0528	.1110	.0002	.1639
hard weight	56	.0161	.1435	.0038	.1634
easy BF	6	.0515	.0969	.0004	.1488
hard BF	8	.0555	.0894	.0003	.1452
pulse compr	8	.0313	.1000	.0151	.1464
CFAR	4	.0777	.0682	-	.1459
throughput	6.7809				
latency	0.7797				

case 3: total number of nodes = 256					Time in seconds
	# nodes	recv	comp	send	total
Parallel read	32	.1041	-	.0004	.1045
Doppler filter	48	.0241	.0453	.0244	.0937
easy weight	12	.0499	.0559	.0002	.1060
hard weight	112	.0319	.0729	.0008	.1056
easy BF	12	.0516	.0486	.0003	.1005
hard BF	16	.0474	.0518	.0003	.0996
pulse compr	16	.0411	.0499	.0079	.0989
CFAR	8	.0643	.0343	-	.0986
throughput	9.9740				
latency	0.3713				

Table 25. Performance results on the Paragon with the I/O implemented as a separate task in which the corner turn and type conversion are embedded in the receive phase.
PFS stripe factor = 64.

case 1: total number of nodes = 64		Time in seconds			
	# nodes	recv	comp	send	total
Parallel read	8	.3242	-	.0004	.3246
Doppler filter	12	.0575	.1742	.0956	.3272
easy weight	3	.1039	.2214	.0002	.3255
hard weight	28	.0375	.2849	.0003	.3227
easy BF	3	.1197	.1921	.0003	.3121
hard BF	4	.1275	.1830	.0002	.3108
pulse compr	4	.0789	.1980	.0296	.3065
CFAR	2	.1693	.1360	-	.3053
throughput		3.3022			
latency		1.2889			

case 2: total number of nodes = 128		Time in seconds			
	# nodes	recv	comp	send	total
Parallel read	16	.1471	-	.0163	.1633
Doppler filter	24	.0048	.1004	.0669	.1722
easy weight	6	.0601	.1109	.0002	.1712
hard weight	56	.0214	.1430	.0059	.1703
easy BF	6	.0524	.0970	.0003	.1497
hard BF	8	.0605	.0895	.0003	.1503
pulse compr	8	.0369	.0994	.0149	.1512
CFAR	4	.0825	.0681	-	.1506
throughput		6.5610			
latency		0.8300			

case 3: total number of nodes = 256		Time in seconds			
	# nodes	recv	comp	send	total
Parallel read	32	.0908	-	.0005	.0913
Doppler filter	48	.0015	.0507	.0244	.0766
easy weight	12	.0434	.0559	.0002	.0995
hard weight	112	.0248	.0727	.0005	.0980
easy BF	12	.0455	.0499	.0003	.0957
hard BF	16	.0390	.0548	.0004	.0942
pulse compr	16	.0349	.0505	.0078	.0932
CFAR	8	.0590	.0342	-	.0932
throughput		10.5710			
latency		0.4629			

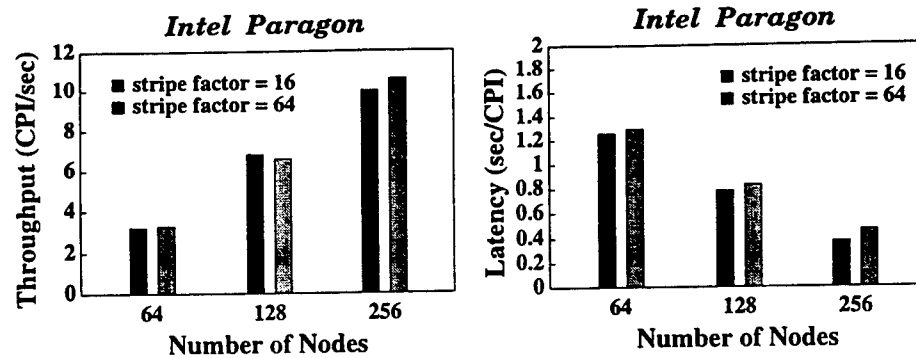


Figure 43. Performance results for the implementation using a separate I/O task in which the corner turn and type conversion are embedded in the receive phase. This figure corresponds to Tables 24 and 25.

Linear speedups were obtained for both throughput and latency.

The performance results for the implementation with the I/O task embedded in the Doppler filter processing task is shown in Tables 26 and 27, for Paragon PFS file systems with 16 and 64 striped directories, respectively. Figure 44 shows the bar charts corresponding to Tables 26 and 27. We observe that the throughput and latency show linear speedups till the case with a total of 120 nodes. The timing for performing read CPI data from disks, corner turn, type conversion, and CPI data redistribution are included in the receive phase of the Doppler filter processing task. When we increase the number of nodes from 32 to 64 in the Doppler filter processing task, the performance of the receive phase does not scale up linearly. This is because of the increasing cost of the all-to-all personalized communication in the sub-CPI data redistribution. The size of each CPI data in our experiments is $128 \cdot 512 \cdot 16 \cdot (2 \cdot 4 \text{ bytes}) = 8\text{M bytes}$. With 64 nodes, the size of data in each send/receive of the all-to-all personalized communication becomes $\frac{8\text{M bytes}}{64 \cdot 64} = 2\text{K bytes}$. In the all-to-all personalized communication, each node has a total of 64 read/receive calls whose communication startup time overwhelms the message transmission time with respect to the relatively small size of the messages (2K bytes each.)

Table 26. Performance results on the Paragon with the I/O implemented in the Doppler filter processing task in which the corner turn and type conversion are embedded in the receive phase.

PFS stripe factor = 16

case 1: total nodes = 31					
Time in seconds					
	node	recv	comp	send	total
Doppler	8	.3188	.2584	.1354	.7127
easy wgt	2	.3794	.3321	.0002	.7118
hard wgt	14	.1446	.5669	.0004	.7119
easy BF	2	.4164	.2865	.0002	.7031
hard BF	2	.3405	.3478	.0002	.6886
PC	2	.2313	.3949	.0583	.6845
CFAR	1	.4121	.2724	-	.6845
throughput	1.4411				
latency	1.9326				

case 3: total nodes = 120					
Time in seconds					
	node	recv	comp	send	total
Doppler	32	.0863	.0660	.0349	.1872
easy wgt	6	.0780	.1110	.0002	.1893
hard wgt	56	.0431	.1429	.0019	.1879
easy BF	6	.0842	.0961	.0003	.1806
hard BF	8	.0886	.0880	.0003	.1770
PC	8	.0616	.0995	.0151	.1763
CFAR	4	.1079	.0683	-	.1762
throughput	5.5923				
latency	0.5047				

case 2: total nodes = 60					
Time in seconds					
	node	recv	comp	send	total
Doppler	16	.1505	.1296	.0681	.3482
easy wgt	3	.1277	.2216	.0002	.3495
hard wgt	28	.0629	.2849	.0003	.3481
easy BF	3	.1419	.1918	.0003	.3340
hard BF	4	.1537	.1756	.0002	.3295
PC	4	.1003	.1985	.0298	.3286
CFAR	2	.1918	.1363	-	.3281
throughput	3.0129				
latency	0.9789				

case 4: total nodes = 238					
Time in seconds					
	node	recv	comp	send	total
Doppler	64	.0625	.0364	.0192	.1181
easy wgt	12	.0675	.0557	.0003	.1234
hard wgt	112	.0494	.0721	.0004	.1219
easy BF	12	.0732	.0482	.0004	.1218
hard BF	14	.0649	.0511	.0003	.1164
PC	16	.0587	.0501	.0078	.1166
CFAR	8	.0821	.0344	-	.1165
throughput	8.4272				
latency	0.2925				

Table 27. Performance results on the Paragon with the I/O implemented in the Doppler filter processing task in which corner turn and type conversion are embedded in the receive phase.

PFS stripe factor = 64

case 1: total nodes = 31					
Time in seconds					
	node	recv	comp	send	total
Doppler	8	.3196	.2586	.1355	.7138
easy wgt	2	.3804	.3321	.0003	.7128
hard wgt	14	.1455	.5670	.0004	.7129
easy BF	2	.4174	.2865	.0002	.7042
hard BF	2	.3413	.3480	.0003	.6896
PC	2	.2321	.3949	.0582	.6852
CFAR	1	.4129	.2724	-	.6852
throughput	1.4390				
latency	1.9368				

case 3: total nodes = 120					
Time in seconds					
	node	recv	comp	send	total
Doppler	32	.0835	.0647	.0455	.1937
easy wgt	6	.0872	.1111	.0002	.1985
hard wgt	56	.0469	.1430	.0074	.1973
easy BF	6	.0934	.0959	.0003	.1896
hard BF	8	.0864	.0895	.0003	.1762
PC	8	.0618	.0998	.0151	.1768
CFAR	4	.1080	.0683	-	.1763
throughput	5.6552				
latency	0.5264				

case 2: total nodes = 60					
Time in seconds					
	node	recv	comp	send	total
Doppler	16	.1504	.1298	.0757	.3558
easy wgt	3	.1341	.2216	.0002	.3559
hard wgt	28	.0697	.2849	.0004	.3550
easy BF	3	.1486	.1913	.0003	.3402
hard BF	4	.1524	.1828	.0002	.3355
PC	4	.1007	.1989	.0317	.3313
CFAR	2	.1918	.1363	-	.3280
throughput	3.0618				
latency	1.0159				

case 4: total nodes = 238					
Time in seconds					
	node	recv	comp	send	total
Doppler	64	.0617	.0327	.0190	.1134
easy wgt	12	.0675	.0558	.0002	.1236
hard wgt	112	.0497	.0724	.0004	.1225
easy BF	12	.0735	.0482	.0003	.1220
hard BF	14	.0652	.0511	.0003	.1166
PC	16	.0590	.0500	.0077	.1167
CFAR	8	.0824	.0343	-	.1167
throughput	8.4237				
latency	0.2927				

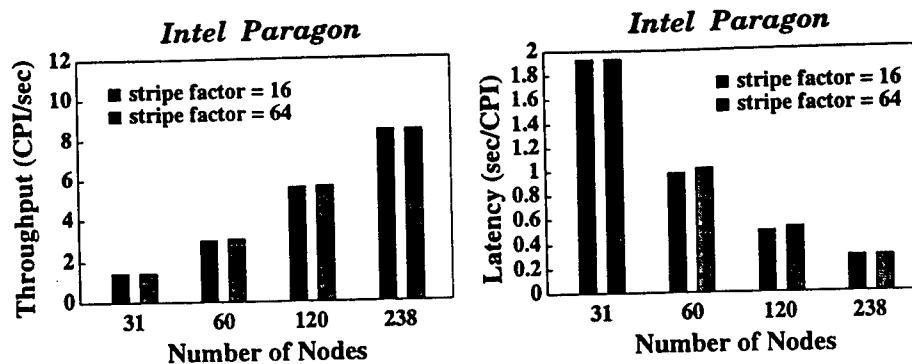


Figure 44. Performance results for the implementation when the parallel I/O, corner turn, and type conversion are embedded in the receive phase of the Doppler filter processing task. This figure corresponds to Tables 26 and 27.

5.6 Summary

In this work, we studied the effects of parallel I/O implementation on the parallel pipeline system for a modified PRI-staggered post-Doppler STAP algorithm. The parallel pipeline STAP system was run portably on Intel Paragon and IBM SP and the overall performance results demonstrated the linear scalability of our parallel pipeline design when the existing parallel file systems were used in the I/O implementations. On the Paragon, we found that a pipeline bottleneck can result when using a parallel file system with a relatively smaller stripe factor. With a larger stripe factor, a parallel file system can deliver higher efficiency of I/O operations and, therefore, improve the throughput performance.

This chapter presented two I/O designs which are incorporated into the parallel pipeline STAP system. One embedded I/O in the original pipeline and the other used a separate I/O task. By comparing the results of these designs, we found that the task structure of the pipeline can be reorganized to further improve the latency. Without adding any compute nodes, we obtained performance improvement in the latency when the last two tasks were combined. We also analyzed the possibility of further improvement by examining the throughput and latency equations.

We also investigated a software approach to implement raw data pre-processing which can often be done by a special purpose hardware. The performance results demonstrate that the parallel pipeline STAP system scaled well even with a more complicated I/O implementation.

Chapter 6

Summary and Conclusions

This dissertation has proposed a parallel pipeline computational model for radar signal processing applications on high performance computers. This model is designed for those radar applications that are computationally intensive and are required to operate in real time. We addressed the advantages of HPC systems over the traditional VLSI based designs in terms of scalability, flexibility, and affordability. In this work, we focused on STAP algorithms which is representative of radar signal processing methods whose parallelization is highly desirable. Based on the computational characteristics of STAP algorithms, the proposed parallel pipeline model captures the computational requirements for this type of application. Although we focused on a specific STAP algorithm and implemented it on the parallel pipeline model throughout the dissertation, this computational model is suitable for the system design of other signal processing applications as well.

In addition to the spatial data dependency, all signal processing applications with space-time relationship show temporal data dependency as well. In the design of the parallel pipeline computational model, these two types of data dependencies are incorporated. The parallelization issues for this model include data redistribution and processor assignment. Data redistribution is divided into two categories: inter-task and intra-task. Inter-task data redistribution occurs when two groups of processors in two different tasks need to exchange data with each other while intra-task data

redistribution involves data transfer among the same group of processors in one task. Both types of data redistribution show two communication patterns: corner turn or left-right shift. In the corner-turn communication pattern, messages need to be packed on the sender side and unpacked on the receiver side. Message packing and unpacking both involve data reorganization whose efficiency depends on the message size and the machine's cache performance.

Processor assignment is one of the most important issues in the implementation of the parallel pipeline model for the embedded real-time signal processing applications. Performance of throughput and latency are two key requirements for this type of applications. Within a single task, the strategy of data partitioning across the processors assigned to this task determines the efficiency of the parallelizations of this task. In the integrated pipeline system, processor assignment affects the overall system performance in terms of throughput and latency. Tradeoffs exist when assigning processors to the tasks such that the throughput is increased and the latency is reduced. These two goal may require two different processor assignments.

A real radar application was used in the implementation of our parallel pipeline computational model. This application is based on a modified PRI-staggered post-Doppler STAP algorithm. AFRL also implemented the same algorithm on a ruggedized version of the Paragon computer by using compute nodes of the machine only as independent resources in a round robin fashion to run different signal data instances. The implemented system had been installed onboard an airborne platform and successfully performed four flight experiments. Using our proposed parallel pipeline model, we implemented the same application on Intel Paragon, IBM SP and SGI Origin. The performance results indicate that our approach scales well both in terms of communication and computation. The throughput and latency results also demonstrate the linear scalability.

Given additional processors, tradeoffs exist between assigning these processors to increase the throughput and to reduce the latency. Throughput of a pipeline system depends on the task with the maximum execution time among all tasks in the pipeline. To improve the throughput, processors should be assigned to the task with

the maximum execution time. On the other hand, latency is determined by the sum of the execution times of the tasks without temporal data dependency. Additional processors can be assigned to those tasks which benefit the most, that is, the tasks whose execution time is reduced the most when more nodes are assigned.

To explore the possibility of further performance improvement, the multi-threaded design of the parallel pipeline STAP system was implemented on the Paragon MP system. Paragon MP system is a massively parallel processing machine with SMP nodes. Each SMP node in the Paragon MP system has three processors sharing the main memory, I/O interface, and other common resources. The thread library implemented on the Paragon uses *POSIX threads* which is not standardized yet. Since the message-passing part of the library is not thread-safe, the multi-threaded design was only implemented in the compute phase of the STAP pipeline system. The approach for using multiple threads is straightforward by dividing the computation load further within each compute node. Performance results indicated that the parallel pipelined implementation scales well for both throughput and latency when the multi-threaded technique is used. Although the concurrent read/write problems limit the multi-threading performance when designing a thread-safe library, our model still provides significant performance improvement by using the Paragon thread-safe numerical libraries.

To study the effect of disk I/O performance on our parallel pipeline model, we incorporated the I/O task designs into the parallel pipeline STAP system. We used the existing parallel file system on the Paragon and SP to perform the read operations of input data to the STAP system. Two I/O task designs were presented in this work. One embedded I/O in the original pipeline and the other used a separate task to perform I/O. We ran the codes using two parallel file systems on the Paragon, each with different sizes of stripe directories. The parallel file system with large stripe directories can deliver higher efficiency of I/O operations to relieve the I/O bottleneck problem in a pipeline system. On the other hand, when comparing the two I/O designs, the throughput results were approximately the same. However, the latency results for the first design were better. This observation leads to the fact

that the task structure of the pipeline can be reorganized to further improve the latency. We combined the last two tasks into a single task and maintained the same number of processors assigned to the whole system and demonstrated this observation. A theoretical analysis was also given by examining the equations of latency and throughput.

A software approach to implement raw signal data pre-processing which can often be done by a special purpose hardware was investigated. The raw data pre-processing involves operations of corner turn and type conversion of a three dimensional data cube. The corner-turn operation in the I/O task can be viewed as a two-phase I/O access strategy. Operation of corner turn also represents the intra-task data dependency in the parallel pipeline system. With a more sophisticated I/O task, the parallel pipeline STAP system scaled well for both throughput and latency.

6.1 Suggestions for Future Work

There are several issues that can be further studied. First, the data redistribution operation between two exclusive groups of processors or even among more than two groups of processors can be further investigated. Since the task structure of a pipeline can be complicated as several processing tasks whose data need to be transferred among several groups of processors, data redistribution can require communication from one group to another group, one group to many groups, many groups to one group, or many groups to many groups of processors. We call this problem a group-to-group data redistribution problem. Development of this type of communication structure is a new research area. In the literature, primitive communication patterns have been well addressed and many optimized approaches have been developed for several architectures. The group-to-group communication structure can be designed based on these primitive communications or a brand new approach needs to be developed.

Optimization of processor assignment in a parallel pipeline system is normally not easily captured by a theoretical analysis. The fact is that the execution time of a task

is determined by its communication and computation time. Even if the computation time of one task can be predicted given a number of processors, the overlapping nature of a pipeline model can hide the communication costs in the other phases of this task, especially when asynchronous send and receive are used with double buffering. Further research on the optimization of processor assignment may first focus on situations with simple communication patterns presented in the inter-task data redistribution, e.g., a parallel pipeline system with only left-right shift communication. Then more general problems can be addressed with more complicated communication patterns.

Since almost all radar applications have real-time constraints, a well designed system should be able to handle any changes in the requirements on the throughput and latency by dynamically allocating or re-allocating processors among tasks. With the capability of interaction with users, the STAP system may need to fine tune some of the signal processing parameters after preliminary detection results are obtained. To design an interactive radar system which is capable of performing processor re-assignment in real time may need to take several issues into consideration, such as overhead of pipeline re-initialization, change of inter-task communication pattern, and so on. This appears to be a fruitful area of research.

Bibliography

- [1] P. Rowe. *COTS Radar and Sonar Systems Solutions*. Multiprocessor Toolsmiths, Inc., Kanata, ON, Canada, 1996.
- [2] J. Ward. Space-Time Adaptive Processing for Airborne Radar. Technical Report 1015, MIT Lincoln Lab., December 1987.
- [3] A. Choudhary and J. Patel. *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems*. Kluwer Academic Publishers, Boston, MA, 1990.
- [4] A. Choudhary and R. Ponnusamy. Run-Time Data Decomposition for Parallel Implementation of Image Processing and Computer Vision Tasks. *Journal of Concurrency, Practice and Experience*, 4(4):313–334, June 1992.
- [5] A. Choudhary and R. Ponnusamy. Parallel Implementation and Evaluation of a Motion Estimation System Algorithm using Several Data Decomposition Strategies. *Journal of Parallel and Distributed Computing*, 14:50–65, January 1992.
- [6] A. Choudhary, B. Narahari, D. Nicol, and R. Simha. Optimal Processor Assignment for Pipeline Computations. *IEEE Trans. on Parallel and Distributed Systems*, April 1994.
- [7] M. Snir and et. al. *MPI The Complete Reference*. The MIT Press, 1995.
- [8] M. Skolnik. *Introduction to Radar Systems, Second Edition*. McGraw-Hill, Inc., New York, 1980.

- [9] D. Taylor and C. Westcott. *Principles of Radar*. Cambridge University Press, New York, 1948.
- [10] J. Lebak, R. Durie, and A. Bojanczyk. Toward A Portable Parallel Library for Space-Time Adaptive Methods. Technical Report CTC96TR242, Cornell Theory Center, June 1996.
- [11] K. Cain, J. Torres, and R. Williams. RT-STAP: Real-Time Space-Time Adaptive Processing Benchmark. Technical Report 96B0000021, MITRE Corporation, February 1997.
- [12] M. Little and W. Berry. Real-Time Multi-Channel Airborne Radar Measurements. In *Proceedings of the IEEE National Radar Conference*, 1997.
- [13] M. Linderman and R. Linderman. Real-Time STAP Demonstration on an Embedded High Performance Computer. *IEEE AES Systems Magazine*, pages 15-21, March 1998.
- [14] S. Olszanskyj, J. Lebak, and A. Bojanczyk. Parallel Algorithms for Space-Time Adaptive Processing. *International Parallel Processing Symposium*, pages 77-81, April 1995.
- [15] Y. Lim and V. Prasanna. Scalable Portable Implementations of Space-Time Adaptive Processing. In *Proceedings of the 10th International Conference on High Performance Computing*, June 1996.
- [16] P. Bhat, Y. Lim, and V. Prasanna. Issues in using Heterogeneous HPC Systems for Embedded Real Time Signal Processing Applications. In *Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, October 1995.
- [17] M. Lee and V. Prasanna. High Throughput-Rate Parallel Algorithms for Space Time Adaptive Processing. *2nd International Workshop on Embedded Systems and Applications*, April 1997.

- [18] D. Martinez. Application of Parallel Processors to Real-Time Sensor Array Processing. *International Parallel Processing Symposium*, April 1999.
- [19] C. Brown, M. Flanzbaum, R. Games, and J. Ramsdell. Real-Time Embedded High Performance Computing: Application Benchmarks. Technical Report MTR94B145, MITRE Corporation, October 1994.
- [20] R. Thakur, A. Choudhary, and J. Ramanujam. Efficient Algorithms for Array Redistribution. *IEEE Trans. on Parallel and Distributed Systems*, 6(7):587–594, June 1996.
- [21] S. Ranka, R. Shankar, and K. Alsabti. Many-to-many Communication With Bounded Traffic. *Symposium on Frontiers of Massively Parallel Computation*, 1995.
- [22] S. Hambruch, F. Hameed, and A. Khokhar. Communication Operations on Coarse Grained Mesh Architectures. *Parallel Computing*, 21:731–751, 1995.
- [23] W. Liu, C. Wang, and V. Prasanna. Portable and Scalable Algorithms for Irregular All-to-All Communication. *16th International Conference on Distributed Computing Systems (ICDCS '96)*, May 1996.
- [24] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*. Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [25] M. Berger and S. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Trans. on Computers*, 36(5):570–580, May 1987.
- [26] F. Berman and L. Snyder. On Mapping Parallel Algorithms into Parallel Architectures. *Journal of Parallel and Distributed Computing*, 4:439–458, 1987.
- [27] M. Linderman and R. Linderman. Real-Time STAP Demonstration on an Embedded High Performance Computer. In *Proceedings of the IEEE National Radar Conference*, 1997.

- [28] R. Brown and R. Linderman. Algorithm Development for an Airborne Real-Time STAP Demonstration. In *Proceedings of the IEEE National Radar Conference*, 1997.
- [29] Intel Corporation. *Paragon System User's Guide*, April 1996.
- [30] Kuck and Associates, Champaign, IL. *CLASSPACK Basic Math Library / C*, 1994.
- [31] A. Choudhary, W. Liao, D. Weiner, P. Varshney, R. Linderman, and M. Linderman. Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers. *International Parallel Processing Symposium*, 1998.
- [32] W. Liao, A. Choudhary, D. Weiner, and P. Varshney. Multi-Threaded Design and Implementation of Parallel Pipelined STAP on Parallel Computers with SMP Nodes. *International Parallel Processing Symposium*, 1999.
- [33] IBM Corp. *IBM AIX Parallel I/O File System: Installation, Administration, and Use*, October 1996.
- [34] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS'93*, pages 56-70, 1993.

Biographical Data

Name: Wei-keng Liao
Date and Place of Birth: June 27, 1966
Taipei, Taiwan
College: National Chung-Hsing University
Taichung, Taiwan
B.S., Applied Mathematics, 1988
Graduate Work: Syracuse University
Syracuse, New York
Research Assistant: 1997-99
Publications:

1. Multi-Threaded Design and Implementation of Parallel Pipelined STAP on Parallel Computers with SMP Nodes. (with A. Choudhary, D. Weiner, and P. Varshney.) *Proceedings of the 13th International Parallel Processing Symposium*, April 1999.
2. Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers. (with A. Choudhary, D. Weiner, P. Varshney, R. Linderman, and M. Linderman.) *Proceedings of the 12th International Parallel Processing Symposium*, April 1998.
3. Dynamic Alignment and Distribution of Irregularly Coupled Data Arrays for Scalable Parallelization of Particle-in-Cell Problems. (with Chao-wei Ou, and Sanjay Ranka.) *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

Appendix D

Report and Users' Manual for the Ozturk Algorithm

1. Introduction

Ozturk algorithm is one of the goodness of fit tests used to check consistency of a null hypothesis against a given sample of random data. For certain real-time applications of the Ozturk algorithm, it is

essential that the processing time is within some acceptable limits. Employing parallel machines seems to be one way of reducing the processing time. Please refer to our final report 'ozturk.ps' for the details of program characteristics, parallelization strategy and final results.

2. How to extract source code?

- A. download ozturk.tar.gz to your local workstation
- B. 'gunzip ozturk.tar.gz' then you will get ozturk.tar
- C. use 'tar xvf ozturk.tar' to extract the archive. It will create a subdirectory new
- D. There are two subdirectories under new, one is
new_hp: directory for the source code running on HP workstations
new_sp: directory for the source code running on IBM SP2

3. How to compile the source code?

First copy the source code to the platform you are going to be running on.

For both SP and HP, there is a single makefile, just type make and you should get the executables respectively.

4. How to run the executables?

On HP workstations, you need to edit a machine list in a file to

indicate which machines are you going to run the code on. such as in the new_hp directory you will find out a text file 'machines', each line corresponds to a HP workstation in my local area network. You need to configure it for workstations in your own local area network.

To run the code on 4 nodes, use 'mpirun' with the following arguments:
' mpirun -np 4 -machinefile machines ozturk '
where, ozturk is executable. The ozturk will be running on the first four machines in machines file.

On IBM SP2, you need to edit a command script, a sample cmd 'ozturk.cmd' requesting 4 nodes is given under the directory new_sp. To run submit the job, use 'llsubmit' command as follows:

```
'llsubmit ozturk.cmd'
```

The results will be directed to 'ozturk.out'.

5. How to configure input file?

Edit 'ozturk.init' under new_hp and new_sp directories respectively.

5. Further questions?

send your email to xhshen@ece.nwu.edu

A Parallel Goodness of Fit Test Algorithm for Realtime Applications

U.Nagaraj Shenoy Alok N. Choudhary Xiaohui Shen Donald Weiner Pramod Varshney

CPDC-TR-9811-022

©1998 Center for Parallel and Distributed Computing
June 1998.

Center for Parallel and Distributed Computing

Department of Electrical & Computer Engineering

Northwestern University

Technological Institute

2145 Sheridan Road, Evanston IL-60208

A Parallel Goodness of Fit Test Algorithm for Real-time Applications*

U.Nagaraj Shenoy Alok N. Choudhary Xiaohui Shen
ECE Department
Northwestern University
Evanston, IL 60208

Donald Weiner Pramod Varshney
EECS Department
Syracuse University
Syracuse, NY 13244

Abstract

Ozturk algorithm is one of the *goodness of fit* tests used to check consistency of a *null hypothesis* against a given sample of random data. For certain real-time applications of the Ozturk algorithm, it is essential that the processing time is within some acceptable limits. Employing parallel machines seems to be one way of reducing the processing time.

In this report we discuss the parallelization of this algorithm on a variety of parallel machines. Our current results indicate that the algorithm scales well up to 6-8 processors depending on the architecture of the parallel machine. Further scaling is impeded by some inherently sequential portions of the algorithm.

1 Introduction

In the analysis of random data, situations are encountered where there may be various statistical models or *hypotheses* that need to be checked against the data. In a typical scenario, one would like to check whether a particular distribution (*the null hypothesis*) consistently represents the data from a certain experiment. Several tests have been proposed for this purpose and the test developed by Professor Aydin Ozturk (commonly known as *Ozturk Algorithm* [1]) is the focus of this report.

Ozturk algorithm is one of the *goodness of fit* tests used to check consistency of a *null hypothesis* against a given sample of random data. Since the algorithm is often used to process real time data, it is necessary to reduce the time needed to process each sample of data. For large sample sizes, it is observed that the computation time of the Ozturk algorithm is not acceptable.

One way to achieve improved processing speed is by employing multiple processors.

*This work was supported by Air Force Materials Command under contract F30602-97-C-0026

The algorithm in the original sequential form exhibits very little parallelism which can be exploited by conventional parallelizing compilers on distributed and distributed shared memory parallel machines. However, there exists significant amount of parallelism which can be exploited by hand parallelizing the code. In the current work we attempt to improve the execution time of this algorithm by employing multiple processors.

As the original sequential code changed later, so we presented two sets of results.

In the next few sections we discuss our initial experience in parallelizing this algorithm on a variety of parallel machines for the original code: IBM SP2 - a 16 processor distributed memory multiprocessor, IBM J30 - an 8 processor symmetric multiprocessor, SGI Origin 2000 - an 8 processor distributed shared memory multiprocessor and a cluster of HP 9000 work-stations. For the new code, the results from SP2 and HP work-stations are presented.

To ensure portability across different parallel architectures we have adopted the message passing style of parallel programming. Further, we have used standard message passing interface namely the MPI [2, 3, 4, 5].

The rest of the paper is organized as follows. In Section 2 we look more closely at the original sequential implementation of the Ozturk algorithm and study how the computation time is distributed among various phases of the algorithm. With this in mind, we evolve a parallelization strategy which is discussed in Section 3. Section 5 describes the code changes from the original sequential code. The results of running our parallel Ozturk algorithm on a variety of parallel machines are presented in Section 4 and 6. Finally we conclude with Section 7.

2 Breakup of computation time

Before deciding the parallelization strategy, the sequential algorithm was profiled to study the breakup of computation time in different phases of the algorithm. The original algorithm which was implemented in FORTRAN has around 50 functions. None of these exhibit significant *loop level parallelism* which can be exploited by conventional parallelizing compilers. Manual parallelization seems to be the only way to parallelize this algorithm.

Among these 50 functions, two top level functions namely `mstar` and `eexpuv` together contribute to more than 80% of the execution time (the function `mstar` along with the functions called by it contributes to more than 50% of the time and `eexpuv` along with the functions called by it contributes to 30% of the time). There also exist other parts of the algorithm, which contribute to roughly 15-20% of the time, that are difficult to parallelize either due to I/O or due to fine granularity.

Our current parallelization strategy mainly concentrates on portions of the algorithm which contribute to roughly 80% of the time.

3 Parallelization Strategy

Our emphasis is on arriving at a portable parallel implementation of the Ozturk algorithm which can run on a variety of parallel machines with little or no modification. We found that a message passing style of parallel programming met this goal. We used the standard message passing interface namely MPI as a portable interface due to its wide availability. The overview of our parallel implementation is shown in Figure 1.

The circles marked by $P_0 \dots P_n$ indicate pro-

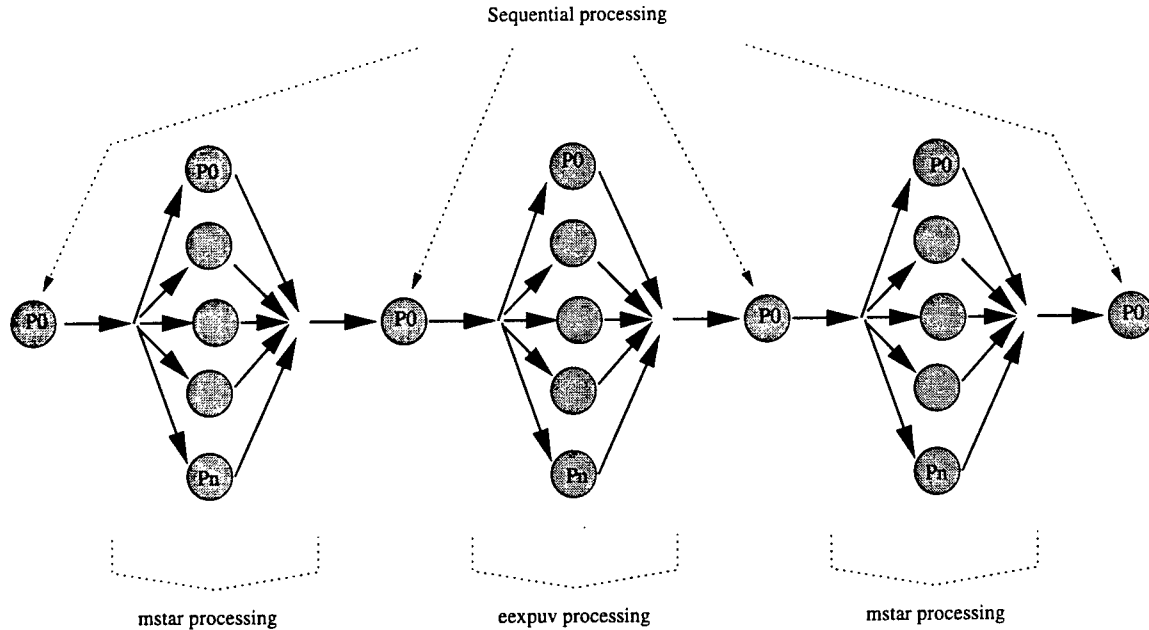


Figure 1: Architecture of parallel Ozturk algorithm

processors $P_0 \dots P_n$. The sequential parts of the algorithm are run on P_0 . The computation in *mstar* and *eexpuv* is shared by all the processors in parallel. Whenever a parallelizable phase of the algorithm is encountered, the computation as well as the data is farmed out to all the processors. These processors work on their portions of the data in parallel and *join* together at the end of the parallel phase. Different MPI primitives like `MPI_BCAST`, `MPI_GATHER` and `MPI_REDUCE` are used to achieve computation partitioning as well as interprocessor communication. The algorithm proceeds as a sequence of alternating sequential and parallel phases.

4 Experimental Results of Old Code

We have ported our parallel Ozturk algorithm on a range of parallel architectures - loosely coupled network of work stations to tightly

coupled shared memory multiprocessors. Table 1 lists these architectures and some of the related numbers.

Three different sizes (100, 250 and 500 points) of input sample data were tried to see how the algorithm scales with sample size. Figure 2 shows the processing time for these samples on different number of processors for these parallel machines.

execution time on SGI Origin 2000 is the least among all the four parallel architectures. This is both because of the higher performance of each processor (195 MHz) as well as better communication bandwidth supported (60 MB/sec) by this machine. The MPI running on Origin is a special implementation of the standard MPI which probably makes best use of the distributed shared memory architecture.

Though the IBM J30 is also a shared memory machine, the performance of the algorithm is not as good as that on Origin primarily because of the lower processor speed (

machine	architecture	processor	clock	comm b/w ¹	nodes	MPI
HP 9000	NOW ²	PA7200	120 MHz	800 KB/sec	8	MPICH
IBM SP2	multicomputer	P2SC	120 MHz	24 MB/sec	16	IBM MPI
SGI Origin 2000	DSM ³	R10000	195 MHz	60 MB/sec	8	SGI MPI
IBM J30	SMP ⁴	PowerPC 604	112 MHz	3 MB/sec	8	MPICH

¹Effective, point to point

²Network of workstations

³Distributed shared memory multiprocessor

⁴Symmetric multiprocessor

Table 1: Platforms on which the Old parallel Ozturk algorithm has been ported

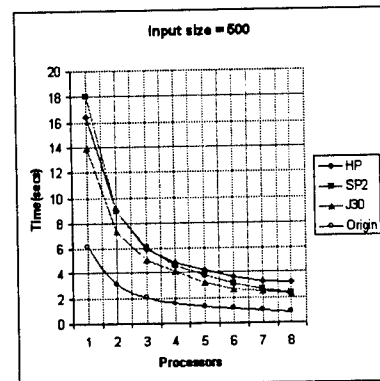
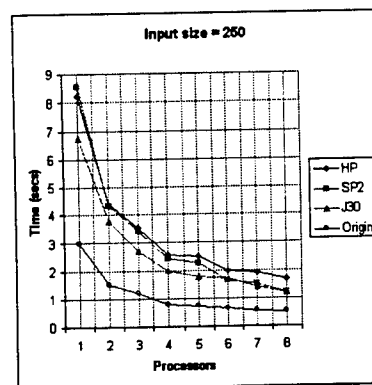
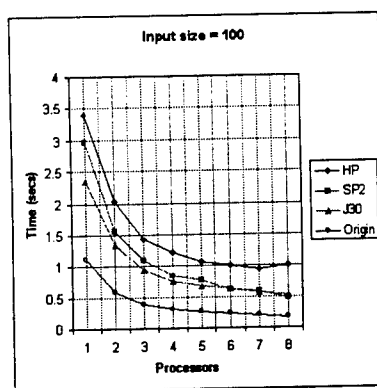


Table 2: Old code: Execution times for different data sizes on various machines

112 MHz) and also the poor communication bandwidth (3 MB/sec). On the other hand, though the SP2 processor is slightly faster (120 MHz) and supports a much higher communication bandwidth (24 MB/sec), J30 always performs slightly better. We suspect that this could be due to I/O overheads since the programs running on SP2 were accessing a remote file system as compared to the programs on J30 which accessed a local file system. Also, the higher communication bandwidth of SP2 may not have given an edge since the algorithm appears to be compute bound rather than communication bound.

Our observation that communication bandwidth does not play a major role in this application stands justified when we compare the performance of the algorithm on SP2 and the network of HP workstations. Architecturally both are distributed memory multiprocessors except that SP2 has a high bandwidth communication fabric (24 MB/sec) as compared to the low speed LAN (800 KB/sec) connecting the HP NOW. For small number of processors, both perform almost identically. The difference shows up for larger number of processors probably because of the reduced concurrency in the communication on the HP NOW.

For small input sizes, only Origin seems to keep up with the near linear speed up. However, SP2 catches up and to some extent scales better than Origin for larger input sizes. Overall, both Origin as well as SP2 seem to scale up very well. Among the machines, HP NOW seems to saturate early.

The speed-up curve seems to taper off beyond 6 to 7 processors in most of the cases. This is understandable given the fact that the parallelized code still has some sequential part which would definitely start dominating as we increase the number of processors irrespective of the architecture.

5 Code Changes

The original sequential code has been changed later. The changes include:

ORIGINAL MODS: (1) added subroutine to read/process an ASCII parameter file (2) removed references to encode/decode; replaced with formatted read/writes

FOLLOW-ON MODS: (1) explicitly declared variable types in the common blocks (2) modified position of '-o' argument in the makefile; make would ignore '-o' option if last argument on the line. (3) removed references to 'structure' keyword in subroutine 'initialize-pgm.f'.

The changes of ozturk code do not affect the parallelization method, so the new parallel code did the same way of parallelization as the old one, i.e. concentrates on mstar and eexpuv functions which take up most of the execution time.

6 Experimental Results of New Parallel Code

We have ported the new parallel Ozturk algorithm on Network of HP workstations and two IBM SP2 machines: one is a 16-node machine at Center for Parallel and Distributed Computing (CPDC) at Northwestern University and one is a 80-node machine at Argonne National Lab. Table 3 lists these machines.

We also tried three different sizes (100, 250 and 500 points) of input sample data to study the scalability. Figure 4 shows the processing time for these samples on different number of processors for these parallel architectures.

Our results show that the execution time on Argonne's SP2 is the best of all parallel architectures and the network of HP workstations is the worst. This is because network of workstations is loosely coupled with poor communication bandwidth. The Ar-

gonne's SP2 has higher communication bandwidth (104Mbytes/s) than CPDC's although they have the same kind of CPU. In addition, the CPDC has to access a remote filesystem when it reads the input, this may introduce some overhead. Therefore, Argonne's SP always outperforms CPDC's SP.

All parallel architectures demonstrate good scalability. The only exception is Network of workstations on the small input size (100), because it's loosely coupled with poor communication bandwidth. But when the input size is large enough, its execution time also decreases appropriately as the number of nodes increases. One observation is that the larger input size, the better speedup can be achieved. Another observation is that the rate of performance gain decreases as the number of node becomes large, this is due to the sequential part of the code may dominate the performance.

Note, the old results can not be compared to the new results since our SP2 and SGI Origin systems have been upgraded a lot and the former results can not be repeated.

7 Conclusion

In this report we discussed the parallelization of the Ozturk algorithm on a range of multiprocessors. The algorithm exhibits significant amount of parallelism needing very little interprocessor communication. We presented the results from our implementation on IBM J30, IBM SP2, SGI Origin 2000 and a network of HP 9000 work-stations for the old code and results on IBM SP2 and network of HP 9000 work-stations for the new code. Our experiments show that the algorithm scales very well for reasonable input sizes and on a small number of processors.

The current algorithm concentrates only on the most compute intensive parts of the sequential algorithm and tries to parallelize

them. The remaining parts are left sequential and are executed by one of the processors (processor 0). This results in the early saturation of the algorithm for larger number of processors since the sequential portions dominate. We observe that some of the sequentiality is introduced by I/O which probably could be eliminated.

References

- [1] Shah Rajiv R. ,*High-Level Adaptive Signal Processing Architecture with Applications to Radar Non-Gaussian Clutter, A New Technique for Distribution Approximation of Random Data*, RL-TR-95-164, Vol II, Rome Laboratory, Sep 1995.
- [2] The MPI Forum, The MPI-Message passing Interface Standard.
- [3] William Gropp, Ewing Lusk and Nathan Doss, A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, *Argonne National Laboratory and Mississippi State University*.
- [4] Silicon Graphics Inc., MPI manual on SGI Origin 2000.
- [5] IBM Corp., MPI manual on IBM SP2.

machine	architecture	processor	clock	comm b/w ¹	nodes	MPI
HP 9000	NOW ²	PA7200	120 MHz	800 KB/sec	8	MPICH
IBM SP2	multicomputer	P2SC	120 MHz	24 MB/sec	16	IBM MPI
IBM SP2(Argonne)	multicomputer	P2SC	120 MHz	104 MB/sec	80	IBM MPI

¹Effective, point to point

²Network of workstations

³Distributed shared memory multiprocessor

⁴Symmetric multiprocessor

Table 3: Platforms on which the New parallel Ozturk algorithm has been ported

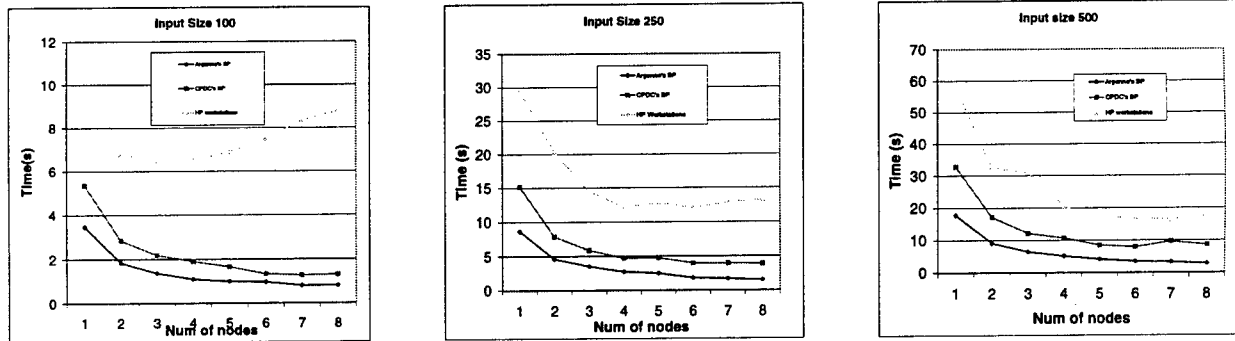


Table 4: New Code: Execution times for different data sizes on various machines

Appendix E

Users' Manual for STAP

User Manual for a Parallel Pipelined PRI-Staggered Post-Doppler STAP Application *

Alok Choudhary

ECE Department

Northwestern University

Evanston, IL 60208

email: choudhar@ece.nwu.edu

Wei-keng Liao,
Donald Weiner, and
Pramod Varshney

EECS Department

Syracuse University

Syracuse, NY 13244

June 9, 1999

*This work was supported by Air Force Materials Command under contract F30602-97-C-0026.

Abstract

This user manual consists of three chapters. The first chapter presents the design and implementation of parallel pipelined Space-Time Adaptive Processing (STAP) algorithms on parallel computers. The second chapter describes the detection performance of STAP algorithm. The third chapter describes the commands of using the parallel pipeline STAP source codes. In the last chapter, default scripts are given for running the parallel pipeline STAP codes on three High Performance Computing (HPC) systems. In particular, the manual describes the issues involved in parallelization, our approach to parallelization and performance results on an Intel Paragon. The process of developing software are also discussed for such an application on parallel computers when latency and throughput are both considered together and presents tradeoffs considered with respect to inter and intra-task communication and data redistribution. The results show that not only scalable performance was achieved for individual component tasks of STAP but linear speedups were obtained for the integrated task performance, both for latency as well as throughput.

Contents

1	Design and Implementation of Parallel Pipelined STAP on Parallel Computers	5
1.1	Algorithm Overview	6
1.2	Model of the parallel pipeline system	8
1.2.1	Parallelization issues and approaches	9
1.3	Design and implementation	11
1.3.1	Doppler filter processing	13
1.3.2	Weight computation	14
1.3.3	Beamforming	17
1.3.4	Pulse compression	19
1.3.5	CFAR processing	19
1.4	Software development	20
2	Detection Performance of STAP algorithm	21
3	Using parallel pipeline STAP commands	25
3.1	Source files	25
3.2	Libraries	26
3.3	Compiling and Linking	27
3.4	Running the program	28
3.4.1	Intel Paragon	28
3.4.2	IBM SP	29
3.4.3	SGI Origin	29

3.5	Input CPI data	30
3.6	Compute node assignment	30
3.7	Parameter file	31
3.8	Description of user specified parameters	32
3.9	Results output	38
3.10	Debugging	39
4	Examples of Running Codes	41
4.1	Parameter files	41
4.2	CPI data files	43
4.3	Compiling	43
4.4	Executing	44
4.5	Output	45
4.6	Script to run with defaults	51

Chapter 1

Design and Implementation of Parallel Pipelined STAP on Parallel Computers

Space-time adaptive processing (STAP) is a well known technique in the area of airborne surveillance radars, which is used to detect weak target returns embedded in strong ground clutter, interference, and receiver noise. Data processing for STAP refers to a 2-dimensional adaptive filtering algorithm which attenuates unwanted signals by placing nulls in the frequency domain with respect to their directions of arrival and/or Doppler frequencies. Most STAP applications are computationally intensive and must operate in real time. High performance computers are becoming mainstream due to the progress made in hardware as well as software support in the last few years. They can satisfy the STAP computational requirements of real-time applications while increasing the flexibility, affordability, and scalability of radar signal processing systems. However, efficient parallelization of STAP, which consists of several different algorithms is challenging, and requires several optimizations.

This manual describes our parallel pipelined implementation of a Pulse Repetition Interval (PRI)-staggered post-Doppler STAP algorithm. The design and implementation of the application is portable. Performance results are presented for the Intel Paragon at the Air Force Research Laboratory (AFRL), Rome, New York. AFRL has successfully implemented this STAP algorithm onboard an airborne platform and performed four flight experiments in May and June 1996 [1]. In that real-time demonstration, live data from a phased array radar was processed by Intel Paragon machine and results showed that high performance computers can deliver a significant performance gain. However, that implementation only

used compute nodes of the machine as independent resources in a round robin fashion to run different instances of STAP (rather than speeding up one instance of STAP.) Using this approach, the throughput may be improved, but the latency is limited by what can be achieved using one compute node. The algorithm consists of the following steps:

For our parallel implementation of this real application we have designed a model of parallel pipeline system where each pipeline is a collection of tasks and each task itself is parallelized. This parallel pipeline model was applied to the STAP algorithm with each step as a task in a pipeline. This permits us to significantly improve latency as well as throughput. For the detail of our implementation and performance results on Intel Paragon, please refer to [2].

1.1 Algorithm Overview

The adaptive algorithm, which cancels Doppler shifted clutter returns as seen by the airborne radar system, is based on a least squares solution to the weight vector problem. This approach has traditionally yielded high clutter rejection, but suffers from severe distortions in the adapted main beam pattern and resulting loss of gain on the target. Our approach introduces a set of constraint equations into the least squares problem which can be weighted proportionally to preserve main beam shape. The algorithm is structured so that multiple receive beams may be formed without changing the matrix of training data. Thus, the adaptive problem can be solved once for all beams which lie within the transmit illumination region. The airborne radar system was programmed to transmit five beams, each 25 degrees in width, spaced 20 degrees apart. Within each transmit beam, six receive beams were formed by the processor.

The algorithm consists of the following steps:

1. Doppler filter processing,
2. Weight computation,
3. Beamforming,
4. Pulse compression, and
5. CFAR processing.

Doppler filtering is performed on each receive channel using weighted Fast Fourier Transforms (FFT's). The analog portion of the receiver compensates the received clutter frequency

to center the clutter frequency at zero regardless of the transmit beam position. This simplifies indexing of Doppler bins for classification as "easy" or "hard" depending on their proximity to mainbeam clutter returns. For the "hard" cases, Doppler processing is performed on two 125-pulse windows of data separated by three pulses (a STAP technique known as "PRI-stagger"). Both sets of Doppler processed data are adaptively weighted in the beamforming process for improved clutter rejection. In the "easy" case, only a single Doppler spectrum is computed. This simpler technique has been termed Post Doppler Adaptive Beamforming and is quite effective at a fraction of the computational cost when the Doppler bin is well separated from mainbeam clutter. In these situations, an angular null placed in the direction of the competing ground clutter provides excellent rejection. Selectable window functions are applied to the data prior to the Doppler FFT's to control sidelobe levels. The selection of a window is a key parameter in that it impacts the leakage of clutter returns across Doppler bins, traded off against the width of the clutter passband.

An efficient method of beamforming using recursive weight updates is made possible by a block update form of the QR decomposition algorithm. This is especially significant in the "hard" Doppler regions, which are computed using separate weights for six consecutive range intervals. The recursive algorithm requires substantially less training data (sample support) for accurate weight computation, as well as providing improved efficiency. Since the hard regions have one sixth the range extent from which to draw data, this approach dealt with the paucity of data by using past looks at the same azimuth, exponentially forgotten, as independent, identically distributed estimates of the clutter to be cancelled. This assumes a reasonable revisit time for each azimuth beam position. During the flight experiments, the five 25 degree transmit beam positions were revisited at a 1-2 Hz rate (5-10 CPIs per second.)

The training data for the easy Doppler regions was selected using a more traditional approach. Here, the entire range extent was available for sample support, so the entire training set was drawn from three preceding CPIs for application to the next CPI in this azimuth beam position. In this case, a regular (non-recursive) QR decomposition is performed on the training data, followed by block update to add in the beam shape constraints.

Pulse compression is a compute intensive task, especially if applied to each receive channel independently. In general, this approach is required for adaptive algorithms which compute different weight sets as a function of radar range. Our algorithm, however, with its mainbeam constraint, preserves phase across range. In fact, the phase of the solution is independent of the clutter nulling equations, and appears only in the constraint equations. The adapted target phase is preserved across range, even though the clutter and adaptive weights may

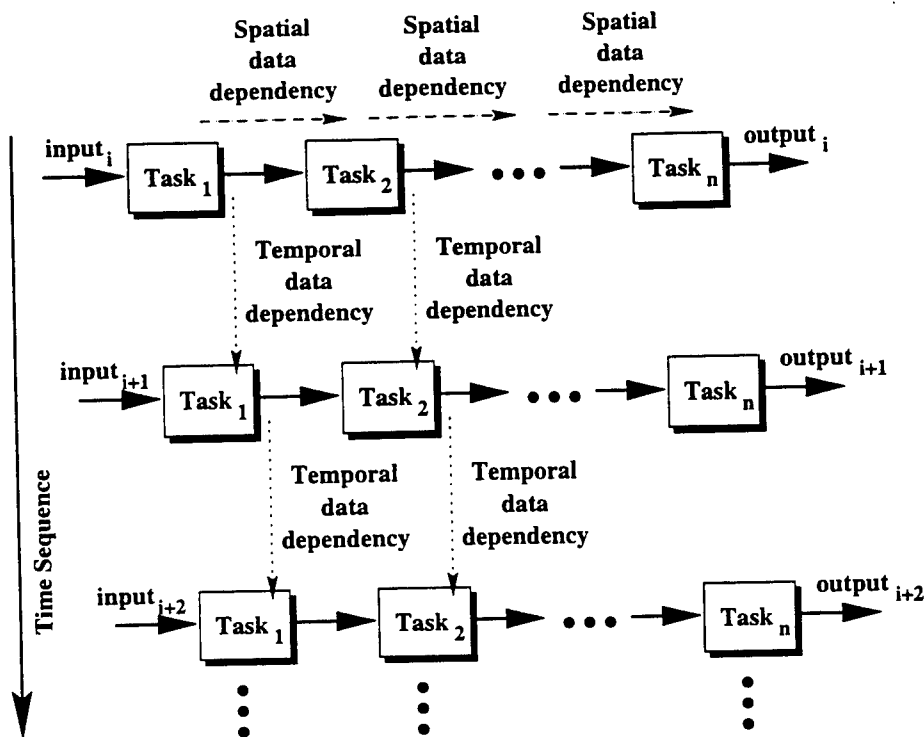


Figure 1.1: *Model of the parallel pipeline system. (Note that $Task_i$ for all input instances is executed on the same number of processors.)*

vary with range. Thus, pulse compression may be performed on the beamformed output of the receive channels providing a substantial savings in computations.

In the sections to follow, we present the process of parallelization and software design considerations including those for portability, task mapping, parallel data redistribution, parallel pipelining and issues involved in measuring performance in implementations when not only the performance of individual tasks is important, but overall performance of the integrated system is critical. We demonstrate the performance and scalability for a large number of processors.

1.2 Model of the parallel pipeline system

The system model for the type of STAP applications considered in this work is shown in Figure 1.1. This model is suitable for the computational characteristics found in these applications. A pipeline is a collection of tasks which are executed sequentially. The input to the first task is obtained normally from sensors or other input devices and the inputs

to the rest of the tasks in the pipeline are the outputs of their previous tasks. The set of pipelines shown in the figure indicates that the same pipeline is repeated on subsequent input data sets. Each block in a pipeline represents one parallel task, which itself is parallelized on multiple (different number of) processors. That is, each task is decomposed into subtasks to be performed in parallel. Therefore, each pipeline is a collection of parallel tasks.

In such a system, there exist both spatial and temporal parallelism that result in two types of data dependencies and flows, namely, spatial data dependency and temporal data dependency [3, 4, 5]. Spatial data dependency can be classified into inter-task data dependency and intra-task data dependency. Intra-task data dependencies arise when a set of subtasks needs to exchange intermediate results during the execution of a parallel task in a pipeline. Inter-task data dependency is due to the transfer and reorganization of data passed onto the next parallel task in the pipeline. Inter-task communication can be communication from the subtasks of the current task to the subtasks of the next task, or collection and reorganization of output data of the current task and then redistribution of the data to the next task. The choice depends on the underlying architecture, mapping of algorithms and input-output relationship between consecutive tasks. Temporal data dependency occurs when some form of output generated by the tasks executed on the previous data set are needed by tasks executing the current data set. We will later see that STAP has both types of data dependencies.

1.2.1 Parallelization issues and approaches

Applications such as STAP entail multiple algorithms (or processing steps), each of which performs particular functions, to be executed in a pipelined fashion. Multiple pipelines need to be executed in a staggered manner to satisfy the throughput requirements. Each task needs to be parallelized for the required performance, which, in turn, requires addressing the issue of data distribution on the subset of processors on which a task is parallelized to obtain good efficiency and incur minimal communication overhead. Given that each task is parallelized, data flow among multiple processors of two or more tasks is required and, therefore, communication scheduling techniques become critical.

Inter-task data redistribution

In an integrated system which implements several tasks that feed data to each other, data redistribution is required when it is fed from one parallel task to another. This is because

the way data is distributed in one task may not be the most appropriate distribution for another task for algorithmic or efficiency reasons. For example, given an input two-dimensional array, one task may process it in a row major fashion. The next task that receives this two-dimensional array may require a column major order. To ensure efficiency of continuity of memory access, data reorganization and redistribution are required in the inter-task communication phase. Data redistribution also allows concentration of communication at the beginning and the end of each task.

We have developed runtime functions and strategies that perform efficient data redistribution [6]. These techniques reduce the communication time by minimizing contention on the communication links as well as by minimizing the overhead of processing for redistribution (which adds to the latency of sending messages). We take advantage of lessons learned from these techniques to implement the parallel pipelined STAP application.

Task scheduling and processor assignment

An important factor in the performance of a parallel system, is how the computational load is mapped onto the processors in the system. Ideally, to achieve maximum parallelism, the load must be evenly distributed across the processors. The problem of statically mapping the workload of a parallel algorithm to processors in a distributed memory system, has been studied under different problem models, such as [7, 8]. The mapping policies are adequate when an application consists of a single task, and the computational load can be determined statically. These static mapping policies do not model applications consisting of a sequence of tasks (algorithms), where the output of one task becomes the input to the next task in the sequence.

Optimal use of resources is particularly important in high-performance embedded applications due to limited resources and other constraints such as desired latency or throughput [9]. When several parallel tasks need to be executed in a pipelined fashion, tradeoffs exist between assigning processors to maximize the overall throughput and assigning processors to minimize a single data set's response time (or latency.) The throughput requirement says that when allocating processors to tasks, it should be guaranteed that all the input data sets will be handled in a timely manner. That is, the processing rate should not fall behind the input data rate. The response time criteria, on the other hand, require minimizing the latency of computation on a particular set of data input.

To reduce the latency, each parallel task must be allocated more processors to reduce its execution time, and consequently, the overall execution time of the integrated system. But

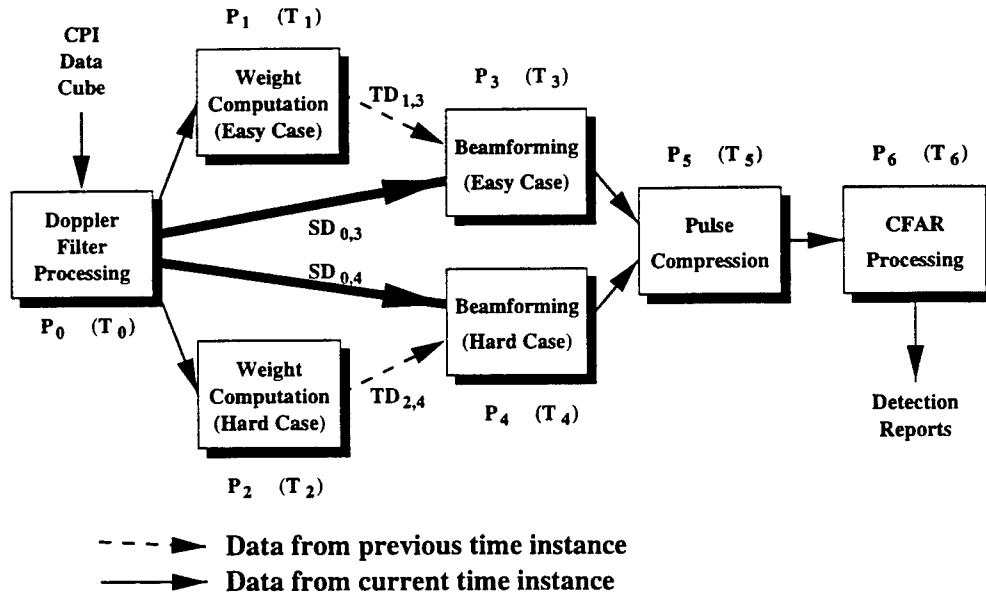


Figure 1.2: Implementation of parallel pipelined STAP. Arrows connecting task blocks represent data transfer between tasks.

it is well known that the efficiency of parallel programs usually decreases as the number of processors is increased. Therefore, the gains in this approach may be incremental. On the other hand, throughput can be increased by increasing the latency of individual tasks by assigning them fewer processors, and therefore, increasing efficiency, but at the same time having multiple streams active concurrently in a staggered manner to satisfy the input-data rate requirements. We next present these tradeoffs and discuss various implementation issues.

1.3 Design and implementation

The design of the parallel pipelined STAP algorithm is shown in Figure 1.2. The parallel pipeline system consists of seven basic tasks. We refer to the parallel pipeline as simply a pipeline in the rest of this paper. The input data set for the pipeline is obtained from a phased array radar and is formed in terms of a coherent processing interval (CPI). Each CPI data set is a 3-dimensional complex data cube comprising of K range cells, J channels, and N pulses. The output of the pipeline is a report on the detection of possible targets. The arrows shown in Figure 1.2 indicate data transfer between tasks. Note that although a single arrow is shown, each represents multiple processors in one task communicating with multiple

processors in another task. Each task i is parallelized by evenly partitioning its work load among P_i processors. The execution time associated with task i , T_i , consists of the time to receive data from the previous task, computation time, and time to send results to the next task.

The calculation of weights is the most computationally intensive part of the STAP algorithm. For the computation of the weight vectors for the current CPI data cube, data cubes from previous CPIs are used as input data. This introduces temporal data dependency. For example, suppose that a set of CPI data cubes entering the pipeline sequentially are denoted by CPI_i , $i = 0, 1, \dots$. At any time instance i , the Doppler filtering task is processing CPI_i and beamforming task is processing CPI_{i-1} . In the meanwhile, the weight computation task is using $CPI_{i-1}, CPI_{i-2}, \dots$ to calculate the weight vectors for CPI_i . The resulted weight vectors will be applied to CPI_i in the beamforming task at next time instance $i + 1$. Thus, temporal data dependencies exist and are represented by arrows with dashed lines, $TD_{1,3}$ and $TD_{2,4}$, in Figure 1.2 where $TD_{i,j}$ represents temporal data dependency of task j on data from task i . In a similar manner, spatial data dependencies $SD_{i,j}$ can be defined and are indicated in Figure 1.2 by arrows with solid lines.

Throughput and latency are two important measures for performance evaluation on a pipeline system. The throughput of our pipeline system is the inverse of the maximum execution time among all tasks, i.e.,

$$throughput = \frac{1}{\max_{0 \leq i \leq 6} T_i}. \quad (1.1)$$

To maximize the throughput, the maximum value of T_i should be minimized. In other words, no task should have an extremely large execution time. With limited number of processors, the processor assignment to different tasks must be made in such a way that the execution time of the task with highest computation time is reduced.

The latency of this pipeline system is the time between the arrival of the CPI data cube at the system input and the time at which the detection report is available at the system output. Therefore, the latency for processing one CPI is the sum of the execution times of all the tasks except weight computation tasks, i.e.,

$$latency = T_0 + \max_{i=3,4} T_i + T_5 + T_6. \quad (1.2)$$

The temporal data dependency does not affect the latency because weight computation tasks use data from the previous instance rather than current CPI. The filtered CPI data cube sent to the beamforming task does not wait for the completion of its weight computation.

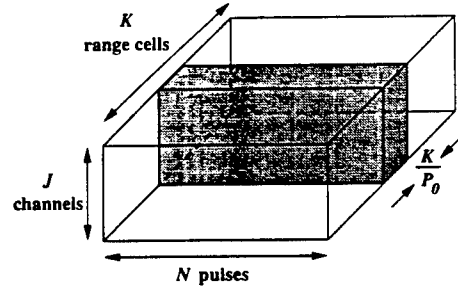


Figure 1.3: *Partitioning strategy for Doppler filter processing task. CPI data cube is partitioned among P_0 processors across dimension K .*

This explains why equation (1.2) does not contain T_1 and T_2 . The overall system latency can be reduced by reducing the execution times of the parallel tasks, e.g., T_0 , T_3 , T_4 , T_5 , and T_6 in our system.

Next, we briefly describe each task and its parallel implementation. A detailed description of the STAP algorithm we used can be found in [10, 11].

1.3.1 Doppler filter processing

The input to the Doppler filter processing task is one CPI complex data cube received from a phased array radar. The computation in this task involves performing range correction for each range cell and the application of a windowing function (e.g. Hanning or Hamming) followed by a N -point FFT for every range cell and channel. The output of the Doppler filter processing task is a 3-dimensional complex data cube of size $K \times 2J \times N$ which is referred to as staggered CPI data. In Figure 1.2, we can see that this output is sent to the weight computation task as well as to the beamforming task.

Both the weight computation and the beamforming tasks are divided into easy and hard parts. These two parts use different portions of staggered CPI data and the associated amounts of computation are also different. Easy weight computation task uses range samples only from the first half of the staggered CPI data while hard weight computation task uses range samples from the entire staggered CPI data. On the other hand, easy and hard beamforming tasks use all range cells rather than some of them. Therefore, the size of data to be transferred to weight computation tasks is different from the size of data to be sent to beamforming tasks. In Figure 1.2, thicker arrows connected from Doppler filter processing task to beamforming tasks indicates that the amount of data sent to the beamforming tasks is more than the amount of data sent to the weight tasks.

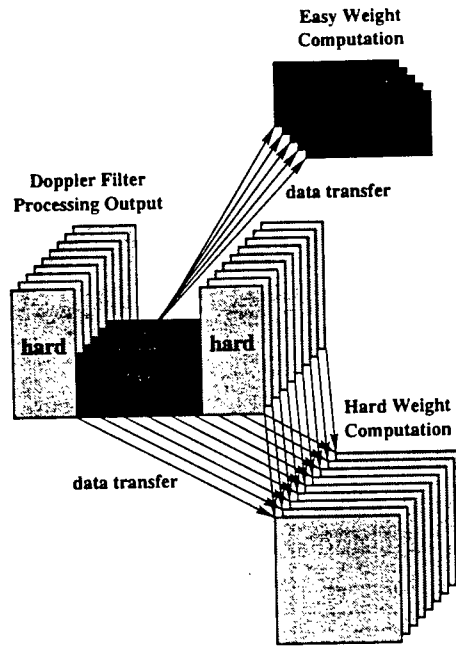


Figure 1.4: *Parallel inter-task communication from Doppler filter processing task to easy and hard weight computation tasks requires different sets of range samples. Data collection needs to be performed before the communication. This can be viewed as irregular data redistribution.*

The basic parallelization technique employed in the Doppler filtering processing task is to partition CPI data cube across the range cells, that is, if P_0 processors are allocated to this task, then each processor is responsible for $\frac{K}{P_0}$ range cells. The reason for partitioning CPI data cube along dimension K is that it maintains an efficient accessing mechanism for continuous memory space. A total of $K \cdot 2J$ N -point FFTs are performed and the best performance is achieved when every N -point FFT accesses its N data sets from a continuous memory space. Figure 1.3 illustrates the parallelization of this step. The inter-task communication from Doppler filter processing task to weight computation tasks is explained in Figure 1.4. Since only subsets of range cells are needed in weight computation tasks, data collection has to be performed on the output data before passing it to the next tasks. Data collection is performed to avoid sending redundant data and hence reduces the communication costs.

1.3.2 Weight computation

The second step in this pipeline is the computation of weights that will be applied to the next CPI. This computation for N pulses is divided into two parts, namely, "easy" and "hard" Doppler bins, as shown in Figure 1.5. The hard Doppler bins (pulses), N_{hard} , are those

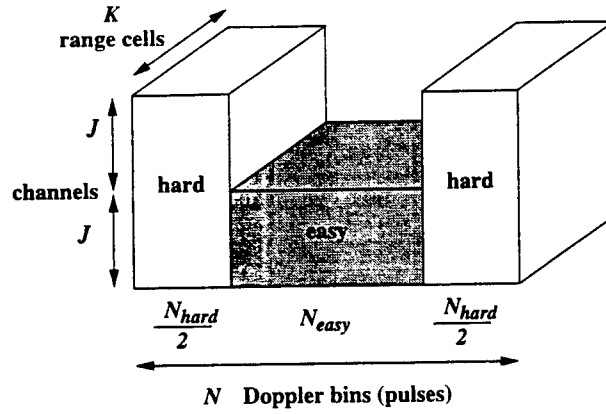


Figure 1.5: *Staggered CPI data partitioned into easy and hard weight computation tasks.*

in which significant ground clutter is expected. The remaining bins are easy Doppler bins, N_{easy} . The main difference between the two is the amount of data used and the amount of computation required. Not all range cells in the staggered CPI are used in weight calculation and different subsets of range samples are used in easy Doppler bins and hard Doppler bins.

To gather range samples for easy Doppler bins to calculate the weight vectors for the current CPI, data is drawn from three preceding CPIs by evenly spacing out over the first one third of K range cells of each of the three CPIs. Easy weight computation task involves N_{easy} QR factorizations, block updates, and back substitutions. In the easy weight calculation, only range samples in the first half of the staggered CPI data are used while hard weight computation employs range samples from the entire staggered CPI. Furthermore, range extent for hard Doppler bins is split into six independent segments to further improve clutter cancelation. To calculate weight vectors for the current CPI, range samples used in hard Doppler bins are taken from the immediately preceding staggered CPI data by evenly spacing out over each of six segment ranges. The hard weight computation task involves $6N_{hard}$ recursive QR updates, block updates, and back substitutions. The easy and hard weight computation tasks process sets of 2-dimensional matrices of different sizes.

Temporal data dependency exists in the weight computation task because both easy and hard Doppler bins use data from previous CPIs to compute the weights for the current CPI. The outputs of this step, the weight vectors, are two 3-dimensional complex data cubes of size $N_{easy} \times J \times M$ and $N_{hard} \times 2J \times M$ for easy and hard weight computation tasks respectively, where M is the number of receive beams. These two weight vectors are to be applied to the current CPI in the beamforming task. Because of the difference sizes of easy and hard weight vectors, beamforming task is also divided into easy and hard parts to handle

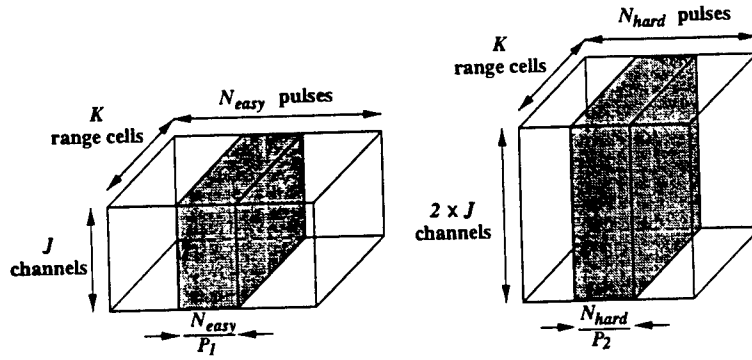


Figure 1.6: *Partitioning strategy for easy and hard weight computation tasks. Data cube is partitioned across dimension N .*

different amounts of computation.

Given the uneven nature of weight computations, different sets of processors are allocated to the easy and hard tasks. In Figure 1.2, P_1 processors are allocated to easy weight computation and P_2 processors to hard weight computation. Since weight vectors are computed for each pulse (Doppler bin), the parallelization in this step involves partitioning of data along dimension N . that is, each processor in easy weight computation task is responsible for $\frac{N_{easy}}{P_1}$ pulses while each processor in hard weight computation task is responsible for $\frac{N_{hard}}{P_2}$ pulses, as shown in Figure 1.6.

Notice that Doppler filter processing and weight computation tasks employ different data partitioning strategies (along different dimensions.) Due to different partitioning strategies, an all-to-all personalized communication scheme is required for data redistribution from Doppler filter processing task to the weight computation task. That is, each of the P_1 and P_2 processors needs to communicate with all P_0 processors allocated to the Doppler filter processing task to receive CPI data. Since only subsets of Doppler filter processing task's output are used in the weight computation task, data collection is performed before inter-task communication. Although data collection reduces inter-task communication cost, it also involves data copying from non-continuous memory space to continuous buffers. Sometimes the cost of data collection may become extremely large due to hardware limitations (e.g. high cache miss ratio.) When sending data to the beamforming task, the weight vectors have already been partitioned along dimension N which is the same as the data partitioning strategy for the beamforming task. Therefore, no data collection is needed when transferring data to the beamforming task.

1.3.3 Beamforming

The third step in this pipeline (which is actually the second step for the current CPI because the result of the weight task is only used in the subsequent time step) is beamforming. The inputs of this task are received from both Doppler filter processing and weight computation tasks, as shown in Figure 1.2. Easy weight vector received from easy weight computation task is applied to the easy Doppler bins of the received CPI data while hard weight vector is applied to hard Doppler bins. The application of weights to CPI data requires matrix-matrix multiplications on two received data sets. Due to different matrix sizes for multiplications in easy and hard beamforming tasks, uneven computational load results. The beamforming task is also divided into easy and hard parts for parallelization purposes. Recall that the weight computation task was divided due to algorithmic reasons. Easy and hard beamforming tasks require different amounts and portions of CPI data, and involve different computational loads.

The inputs for the easy beamforming task are two 3-dimensional complex data cubes. One data cube which is received from the easy weight computation task is of size $N_{easy} \times M \times J$. The other is from Doppler filter processing task and its size is $N_{easy} \times J \times K$. A total of N_{easy} matrix-matrix multiplications are performed where each multiplication involved two matrices of size $M \times J$ and $J \times K$ respectively. Hard beamforming task also has two input data cubes which are received from Doppler filter processing and hard weight computation tasks. The data cube of size $6N_{hard} \times M \times 2J$ is received from hard weight computation task and the Doppler filtered CPI data cube is of size $N_{hard} \times 2J \times K$. Since range cells are divided into 6 range segments, there are a total of $6N_{hard}$ matrix-matrix multiplications in hard beamforming. The results of the beamforming task are two 3-dimensional complex data cubes of size $N_{easy} \times M \times K$ and $N_{hard} \times M \times K$ corresponding to easy and hard parts respectively.

In a manner similar to the weight computation task, parallelization in this step also involves partitioning of data across the N dimension (Doppler bins.) Different sets of processors are allocated to easy and hard beamforming tasks. Since the cost of matrix multiplications can be determined accurately, the computations are equally divided among the allocated processors for this task. As seen from Figure 1.2, this task requires data to be communicated from the first as well as the second task. Because data is partitioned along different dimensions, an all-to-all personalized communication is required for data redistribution between Doppler filter processing and beamforming tasks. The output of Doppler filter processing task is a data cube of size $K \times 2J \times N$ which is redistributed to the beamforming task after

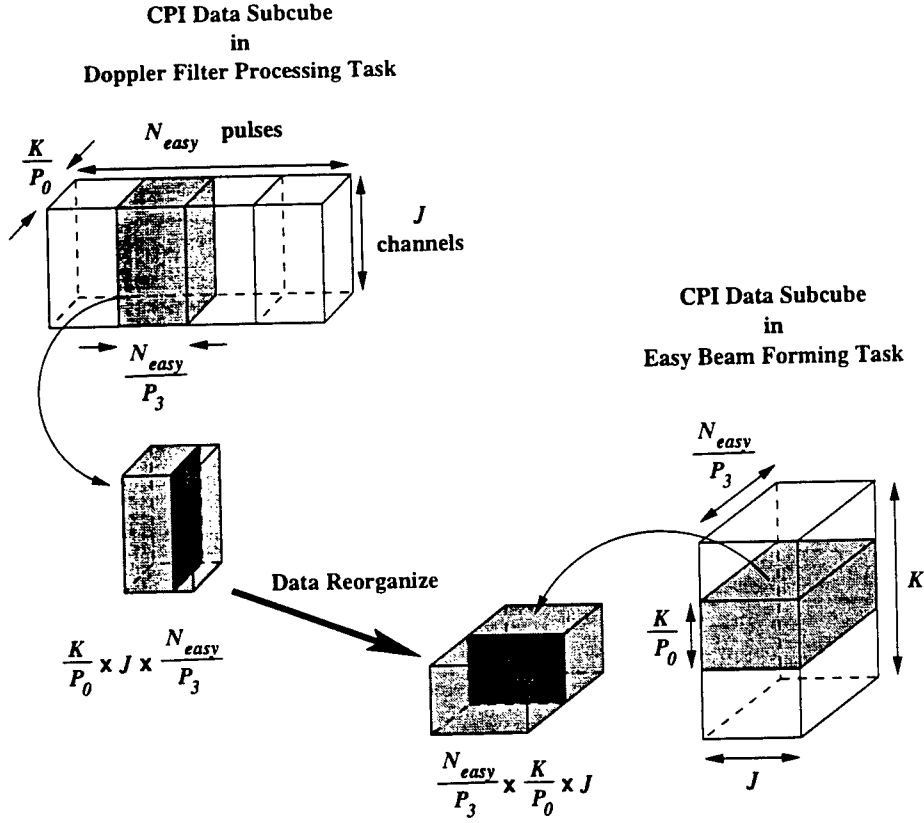


Figure 1.7: Data redistribution from Doppler filter processing task to easy beamforming task. CPI data subcube of size $\frac{K}{P_0} \times J \times \frac{N_{easy}}{P_3}$ is reorganized to subcube of size $\frac{N_{easy}}{P_3} \times \frac{K}{P_0} \times J$ before sending from one processor in Doppler filter processing task to another in easy beamforming task.

data reorganization in the order of $N \times K \times 2J$. Data reorganization has to be done before the inter-task communication between the two tasks takes place, as shown in Figure 1.7.

Data reorganization involves data copying from non-continuous memory space and its cost may become extremely large due to cache misses. For example, two Doppler bins in the same range cell and the same channel are stored in contiguous memory space. After data reorganization, they are $\frac{K}{P_0} \cdot J$ element distance apart. Therefore, if P_0 is small and the size of CPI data subcube partitioned in each processor is large then it is quite likely that expensive data reorganization will be needed which becomes major part of communication overhead. The algorithms which perform data collection and reorganization are crucial to exploit the available parallelism. Note that receiving data from weight computation tasks does not involve data reorganization or data collection because they have the same partitioning strategy (along dimension N .)

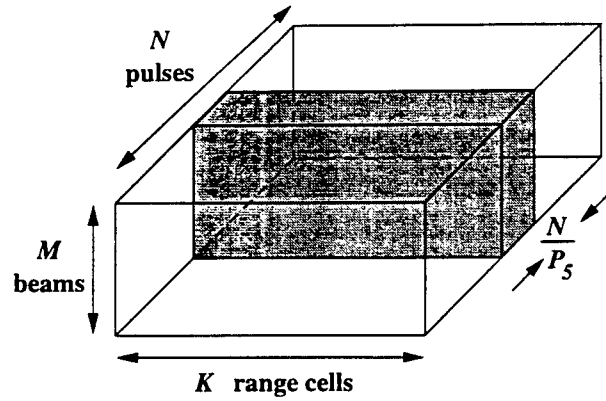


Figure 1.8: *Partitioning strategy for pulse compression task. Data cube is partitioned across dimension N into P_5 processors.*

1.3.4 Pulse compression

The input to the pulse compression task is a 3-dimensional complex data cube of size $N \times M \times K$. This data cube consists of two subcubes of size $N_{easy} \times M \times K$ and $N_{hard} \times M \times K$ which are received from easy and hard beamforming tasks respectively. Pulse compression involves convolution of the received signal with a replica of the transmit pulse waveform. This is accomplished by first performing K -point FFTs on the two inputs, point-wise multiplication of the intermediate result and then computing the inverse FFT. The output of this step is a 3-dimensional real data cube of size $N \times M \times K$. The parallelization of this step is straightforward and involves the partitioning of data cube across the N dimension. Each of the FFTs could be performed on an individual processor and hence each processor in this task gets an equal amount of computation. Partitioning along the N dimension also results in an efficient accessing mechanism for continuous memory space when running FFTs. Since both beamforming and pulse compression tasks use the same data partitioning strategy (along dimension N), no data collection or reorganization is needed prior to communication between these two tasks.

1.3.5 CFAR processing

The input to this task is an $N \times M \times K$ real data cube received from the pulse compression task. The sliding window constant false alarm rate (CFAR) processing compares the value of a test cell at a given range to the average of a set of reference cells around it times a probability of false alarm factor. This step involves summing up a number of range cells

on each side of the cell under test, multiplying the sum by a constant, and comparing the product to the value of the cell under test. The output of this task, which appears at the pipeline output, is a list of targets at specified ranges, Doppler frequencies, and look directions. The parallelization strategy for this step is the same as for the pulse compression task. Both tasks partition data cube along the N dimension. Also, no data collection or reorganization is needed in pulse compression task before sending data to this task.

1.4 Software development

All the parallel programs development and their integration is being performed using C language and message passing interface (MPI) [12]. All the functions needed for data redistribution etc. are also being developed in the same fashion. This permits easy portability across various platforms which support C language and MPI. Since MPI is becoming a de facto standard for high-performance systems, we believe the software will be portable. To facilitate upward or downward scalability, the number of processors, data sizes and other important parameters are runtime inputs so that the same program can be run on different number of processors without compiling it again. This allows, for example, the same function to be executed on 2, 4 and so on, number of processors.

Chapter 2

Detection Performance of STAP algorithm

The algorithm which has been parallelized is a PRI-staggered post-Doppler STAP algorithm. Assume the data cube consists of N pulses, J channels, and K range cells. With this algorithm, Doppler filtering is first performed separately on the N pulses received in each channel. In effect, this produces at each channel the output of N Doppler filters which subdivide the Doppler frequency interval into N contiguous Doppler bins. The intention is that each Doppler filter is designed to have suitably low side lobes such that it rejects all of the clutter with Doppler frequencies outside of the filter passband. In this way, the residual clutter along the clutter ridge is localized in term of its spatial frequencies. Adaptive spatial filtering is subsequently performed to reduce the residual clutter.

The philosophy of the post-Doppler STAP algorithm is illustrated in the Figure 2.2. Let

- d = inter-channel spacing
- f_r = pulse repetition frequency
- λ_0 = wavelength of transmitted carrier frequency
- v_a = velocity of airborne platform in χ -direction
- θ = elevation angle
- ϕ = azimuth angle

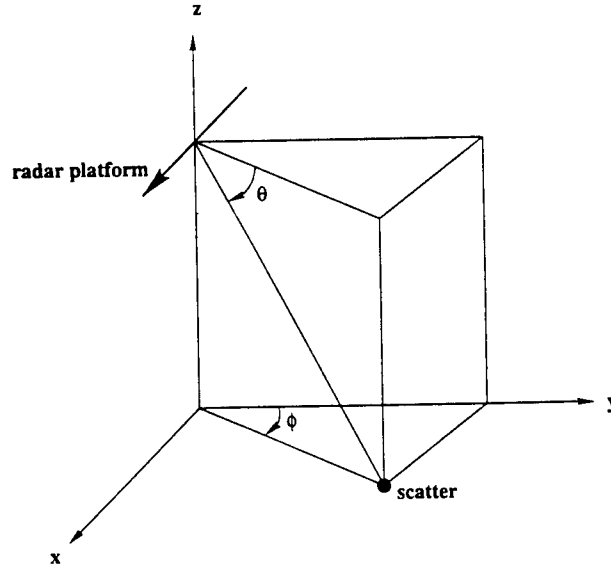


Figure 2.1: Radar platform - array in x - z plane parallel to x axis. θ = elevation angle and ϕ = azimuth angle.

Assuming no velocity misalignment (i.e. zero crab angle), the clutter return from a point scattered in the (θ, ϕ) direction has a normalized Doppler frequency given by

$$\bar{\omega} = \frac{2v_a}{f_r \lambda_0} \cos \theta \sin \phi \quad (2.1)$$

and the normalized spatial frequency equals to

$$\nu = \frac{d}{\lambda_0} \cos \theta \sin \phi. \quad (2.2)$$

It follows that the clutter returns appear upon the clutter ridge specified by

$$\bar{\omega} = \beta \nu \quad (2.3)$$

where

$$\beta = \frac{2v_a}{f_r d}. \quad (2.4)$$

The Figure 2.2 shows the clutter ridge corresponding to $\beta = 1$. Ideally, the Doppler filter rejects all of the clutter outside its passband (the shaded area in the Figure). The spatial frequencies of the residual clutter are then localized to the shaded region shown in the Figure 2.2. Adaptive spatial filtering is then used to remove the residual clutter. A target within the Doppler filter will be easy to detect when it is located far from the clutter ridge. The

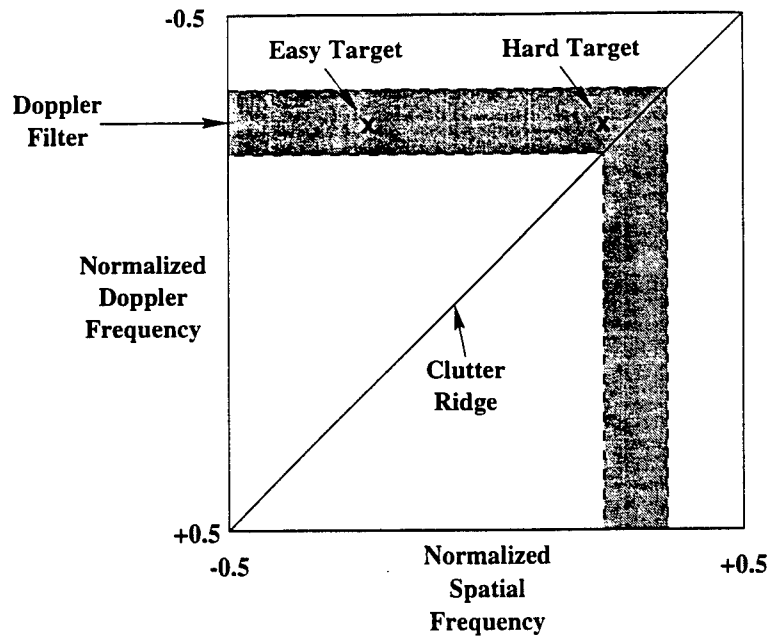


Figure 2.2: *Philosophy of the post-Doppler STAP algorithm.*

target becomes more difficult to detect as it moves toward the clutter ridge and it may not be possible to detect the target when it is too close to the clutter ridge.

When the side lobes of the Doppler filter are not sufficiently low, the strong clutter in the main beam of the radar transmitter will not be completely rejected by the Doppler filter. The adaptive spatial filter will then attempt to remove this clutter as well. In the process, the signal power received from the target may be reduced making it more difficult to detect the target.

The problem becomes more difficult when frequency aliasing occurs. This will be the case when $\beta > 1$. Then more than one clutter ridge will appear within the passband of the Doppler filter. As a result, the target is more likely to be close to a clutter ridge and, therefore, more difficult to detect. Additional clutter ridges may appear within the passband of the Doppler filter when there is velocity misalignment (i.e., a nonzero crab angle). The clutter ridge then expands into an ellipse which can be troublesome when there is significant antenna gain in the backward looking direction. Frequency aliasing aggravates the problem.

Because the clutter covariance matrix is unknown, it must be estimated. Typically, this is done using secondary (or training) data from nearby range cells. In a severely non-homogeneous environment, the secondary data will not be representative of the clutter data in the range cell under test. As a consequence, the spatial adaptive filter may do a poor job

of cancelling the localized clutter along the clutter ridge and a target may not be detected.

Chapter 3

Using parallel pipeline STAP commands

In this chapter, we describe the parallel pipeline STAP source file structures and explain the commands to perform the following tasks on various HPC system platforms:

- compiling and linking application,
- running application,
- input data files, and
- output data files.

The target machines for current implementations are Intel Paragon, IBM SP, and SGI Origin. Besides, the source codes can also be ported to other machines with modification on linked libraries.

3.1 Source files

All source files are written in standard ANSI C language. Several make files have been incorporated with the source files: `Makefile`, `Makefile.common`, `Makefile.paragon`, `Makefile.sp`, and `Makefile.sgi`. According to the make file, `Makefile.common`, source files are generally consist of 8 groups:

1. Doppler filter processing (`FILTER_GRP`),

Table 3.1: *Source file names for each of 7 tasks.*

task	sending phase	computation phase	receiving phase
Doppler filter	random_gen_cpi.c	rawToFFT.c	filter_grp_send.c
easy weight	wt_easy_grp_recv.c	easy_dop.c	wt_easy_grp_send.c
hard weight	wt_hard_grp_recv.c	hard_dop.c	wt_hard_grp_send.c
easy beamforming	bf_easy_grp_recv.c	beamforming_easy.c	bf_easy_grp_send.c
hard beamforming	bf_hard_grp_recv.c	beamforming_hard.c	bf_hard_grp_send.c
pulse compression	pc_grp_recv.c	pulse_compress.c	pc_grp_send.c
CFAR	cfar_grp_recv.c	cfar_detection.c	-

2. easy weight computation (WT_EASY_GRP),
3. hard weight computation (WT_HARD_GRP),
4. easy beamforming (BF_EASY_GRP),
5. hard beamforming (BF_HARD_GRP),
6. pulse compression (PC_GRP),
7. CFAR processing (CFAR_GRP), and
8. utility subroutines.

3.2 Libraries

Libraries used in the source codes include Message Passing Interface (MPI) [12], standard ANSI C math library, Basic Linear Algebra Subroutines (BLAS) library, and Fast Fourier Transform(FFT) library. Since most of the HPC platforms support ANSI C language and MPI, the part of source codes linking with MPI and ANSI C math libraries are portable. The header file for using MPI library is `mpi.h` and for using ANSI C math library is `math.h`. The libraries linked are `libmpi.a` and `libm.a` or `libmpi.so` and `libm.so`. On the other hand, there are various implementations of the BLAS and FFT libraries on different system platforms. The following is the list of BLAS and FFT libraries implemented on the HPC system platforms that parallel pipeline STAP source codes have been ported successfully.

- Intel Paragon running OSF/1 operating system

- CLASSPACK Basic Math Library [13],
- CLASSPACK Signal Processing Library [14].
- header file: `kai.c.h`.
- library files: `libkmath.a` and `libksignal.a`.
- IBM SP running AIX operating system
 - Engineering and Scientific Subroutine Library (ESSL) [15],
 - header file: `essl.h`.
 - library file: `libessl.a`.
- SGI Origin running IRIX 6.4 or 6.5 operating system running MIPS R10000 CPUs
 - SGI/Cray Scientific Library (SCSL) [16],
 - library file: `libscs.so`.

3.3 Compiling and Linking

Make files are available for compiling the source codes on different HPC platforms:

- `Makefile`,
- `Makefile.common`,
- `Makefile.paragon`,
- `Makefile.sp`, and
- `Makefile.sgi`.

Since the compiling and linking environments differ among target HPC systems, the make file with extension, `paragon`, `sp`, and `sgi` are for Intel Paragon, IBM SP, and SGI Origin respectively. To compile the source codes on Intel Paragon, use the command

```
% make -f Makefile.paragon
```

To compile the source codes on IBM SP, use the command

```
% make -f Makefile.sp
```

To compile the source codes on SGI Origin, use the command

```
% make -f Makefile.sgi
```

The resulted executable file is `main.f`.

Users can also use other C compiler options by modifying the macro `CFLAGS` in `Makefile.common`. In `Makefile.common`, two macros defined for compiling are

```
DFLAGS    = -DFLOAT_TYPE
CFLAGS     = -O2
```

The macro `CFLAGS` specifies compiling options. A flag `-O2` is given to specify the level of optimization for the object code is 2. For example, users can change it to `-g` for debugging purpose. Please see the online manual of C compiler for detail of all flags on the system platform used. Macro `DFLAGS` is a list for define macros for compiling. A defined macro `FLOAT_TYPE` shown here makes the application compiled with all real numbers declared in the source program using `float` type (single precision). If this variable list is given empty, i.e.

```
DFLAGS     =
```

the application is compiled with all real numbers using `double` type (double precision).

3.4 Running the program

The program execution environments also differ among target HPC systems, We now explain how to run the program on Intel Paragon, IBM SP, and SGI Origin.

3.4.1 Intel Paragon

To run in interactive mode, a partition of compute nodes must first be made. Usually, partition of interactive mode is made under `.compute` partition. By using command `"lspart -r ."`, a list available partitions is shown such that users can reserve a group of compute nodes under desired partition tree.

```
% lspart -r .
```

USER	GROUP	ACCESS	SIZE	FREE	RQ	EPL	PARTITION
..							
root	daemon	754	10	10	SPS	5	io
root	daemon	754	6	6	SPS	5	service
root	daemon	777	28	28	SPS	5	sunmos
root	daemon	777	1	1	SPS	5	atm_hippi
root	daemon	766	30	30	SPS	5	ditp
root	daemon	744	232	232	SPS	5	compute
.sunmos:							
root	daemon	777	28	28	SPS	5	interactive
.compute:							
root	daemon	766	85	85	SPS	5	OPEN
.compute.OPEN:							
pottsg	AFIT	766	4	4	SPS	5	test
wkliao	PAHPES	766	16	16	SPS	5	stap

For example, to make a partition with 16 compute nodes under .compute partition, a user can use the command

```
% mkpart -sz 16 OPEN.stap
```

For more information of making partition, please refer *Paragon User's Guide* [17]. A command line to execute the application interactively can be

```
% main_f -sz 16 -pn OPEN.stap.
```

3.4.2 IBM SP

To run in interactive mode, a command "mpirun" is for launching an MPI application. A command can be as simple as

```
% mpirun -np 16 main_f
```

3.4.3 SGI Origin

To run in interactive mode, the command "mpirun" is also available on SGI Origin for launching an MPI application. A command can also be as simple as


```
% mpirun -np 16 main_f
```

3.5 Input CPI data

The input CPI data sets for this parallel pipeline STAP application are generated by a C random number generator, `random()`. Every processor in Doppler filter processing group calls the function `random_gen_cpi()` in file `random_gen_cpi.c` with its processor id as the random seed. Therefore, when the number of processors assigned to Doppler filter processing group differs, the overall CPI data differs. If users want to use real CPI data as inputs, please refer to the Debugging section to set the debug flag in `filter_grp.c`.

3.6 Compute node assignment

The file `proc.dat` controls the compute node allocations on the program. An example of this file is:

```
# processor numbers assigned for each group
32 16 112 16 28 16 16
16 8 56 8 14 8 8
8 4 28 4 7 4 4
```

where the comments followed by `#` in each line are ignored. In this example, the program will execute three times, each with different numbers of compute nodes assigned to each of the 7 groups of tasks. That is, the first line in this example will assign

- 32 compute nodes to `FILTER_GRP`,
- 16 compute nodes to `WT_EASY_GRP`,
- 112 compute nodes to `WT_HARD_GRP`,
- 16 compute nodes to `BF_EASY_GRP`,
- 28 compute nodes to `BF_HARD_GRP`,
- 16 compute nodes to `PC_GRP`, and
- 16 compute nodes to `CFAR_GRP`,

and so forth. Before running the program, users should allocate sufficient number of compute nodes. This is, the number of reserved compute nodes should be at least greater than the total number specified in the file `proc.dat`. In the above example, 236 the the smallest number needed to run the program.

3.7 Parameter file

The parameter file for the parallel pipeline STAP application is `param.dat`. The program reads this file and checks the ranges of the given parameters before it actually runs. An example of `param.dat` is

```
-k      512      # number of range cells
-j      16      # number of channels
-n     128      # number of pulses
-r       3      # number of reference CPIs
-m     24      # total number of CPIs (besides the reference CPIs)
-p       3      # number of zero padding
-w Hanning      # windowing function: Hanning or Hamming
-h     56      # number of hard Doppler bins
-e     26      # number of range samples for easy weight
-u     0.3333   # fraction of range cells for extracting easy weight samples
-s     39      # number of range samples for hard weight
-g       6      # number of segments for each hard Doppler bin
-l       5      # number of broad transmit beams
-d       2      # broad transmit beams direction
-b       6      # number of receive beams for each broad transmit beam
-V     SVs     # filename of the steering vector (in Matlabs 4.0)
-c       0.5    # beam constraint weight
-f       0.05   # frequency constraint weight
-o       0.6    # forgetting factor
-C replica      # filename for replica array used in pulse compression
-a       2      # number of guard cells for the sliding window
-i      10      # number of range cells for the window size
-q      12.7    # false alarm factor
-v     0.0001   # probability of false alarm for order statistic CFAR
-y       0.0    # guessing left boundary root of solving threshold equation
-z     100.0    # guessing right boundary root of solving threshold equation
```

```

-x      0.0001 # accuracy of bisection root finding for solving threshold
-t      17      # order number for order statistic CFAR
-R      17400    # recording start range (in meters)
-S      1.0E6    # A/D sampling frequency (in Hz)
-N      16      # number of bits representing one CPI element
-P      61.1E-6 # transmit pulse width (in seconds)
-F      450.0E6 # transmit frequency (in Hz)
-B      0.5E6    # transmit bandwidth (in Hz)
-D      0.333    # azimuth element spacing (in meters)
-A      90      # mechanical boresight azimuth (degree)
-E      3       # mechanical boresight elevation (degree)

```

where the comments followed by # are ignored.

3.8 Description of user specified parameters

A description of user specified parameters are given as follows.

-k *value* number of range cells for one CPI data cube.

-j *value* number of channels for one CPI data cube.

-n *value* number of pulses for one CPI data cube.

-m *value* total number of CPIs (besides the reference CPIs). This number should be at least one. The measured timing results are obtained by recording the process time of the middle CPIs. The first two and last two CPI process time do not count.

-r *value* number of previous reference CPIs for weight computation tasks. Reference CPIs are used to calculate the adaptive weight vectors that is applied to the current CPI in beamforming task. The suggested default value is 3.

-p *value* number of zeros padded for each Doppler FFT. For example, given 128 pulses, if 3 zero padding and 2 staggers are chosen then a set of FFTs is performed on the first 125 pulses and another set of FFTs performed on the last 125 pulses. In each case the last 3 samples are padded with zeros. This value depends on the number of pulses. The default values is 3.

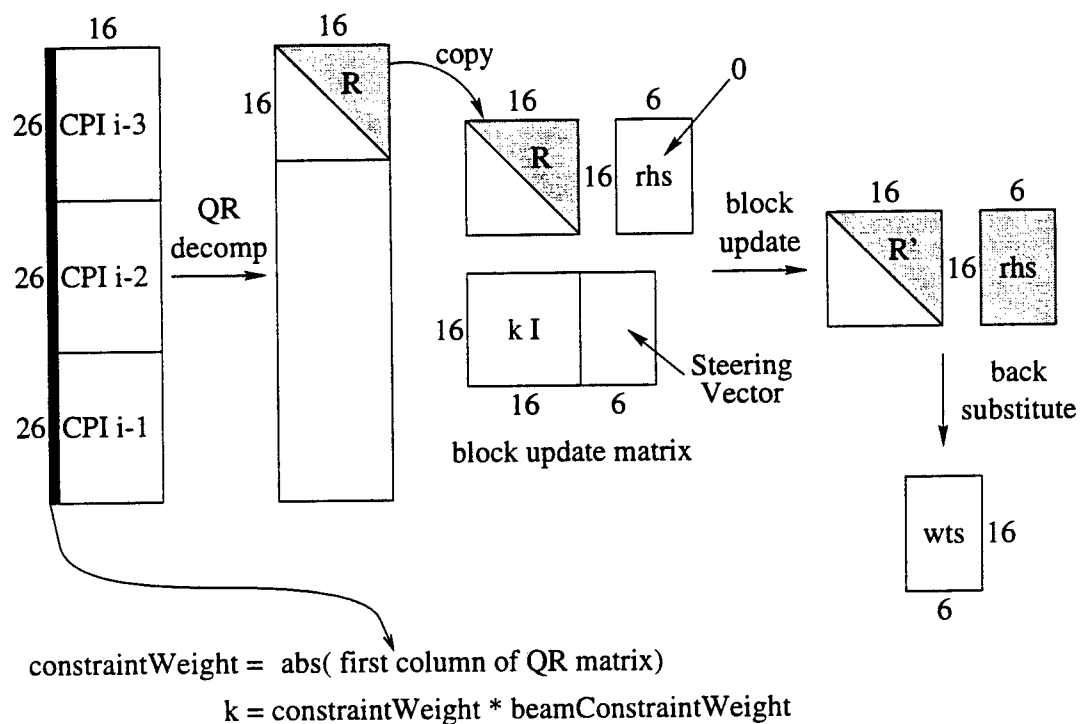
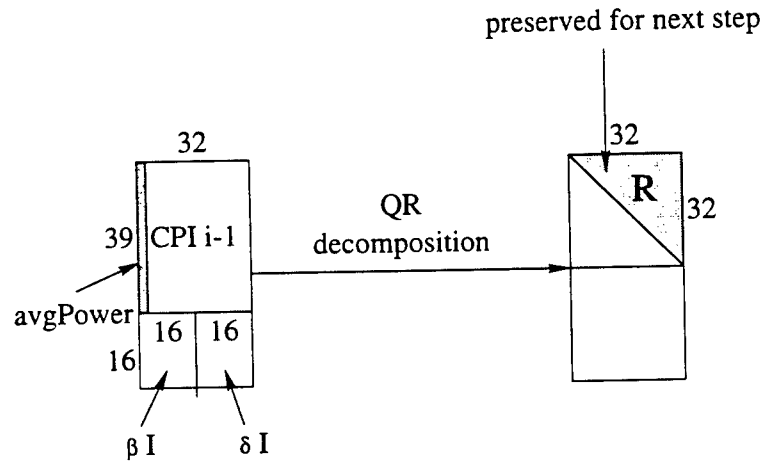


Figure 3.1: Weight vector calculation for one easy Doppler bin, given 16 channels, 26 easy range samples, 3 reference CPIs, and 6 receive beams.



avgPower : $\text{abs}(\text{first column of QR matrix}) / 39$

α : from DFT_matrix[6][128] in stagger_dft.h

$$\alpha = \omega^{jk}$$

β : (freqConstraintWeight * avgPower, 0)

δ : freqConstraintWeight * avgPower * $-\alpha$

Figure 3.2: Weight vector calculation for one hard Doppler bin (Initial step), given 2 staggers. 16 channels, 39 range samples, and 6 receive beams.

- w *value* windowing coefficient name for Doppler filter processing on each staggered CPI. It can be Hanning or Hamming windows. The default windowing coefficient is Hanning window.
- h *value* number of Doppler bins for hard case of adaptive weight computation. The suggested default value is $(pulses \cdot \frac{56}{128})$.
- e *value* number of range cell samples for easy weight computation. This refers to the number of range samples from each reference CPI to be used in the matrix on which a QR factorization is performed for easy weight computation. Figure 3.1 shows the QR factorization used in easy weight computation, given 16 channels and 3 reference CPIs. The suggested default value is $\frac{5 \cdot \text{number of channels}}{\text{number of reference CPIs}}$.
- u *value* fraction of range cells for extracting sample range cells for easy weight computation. The suggested default value is $\frac{1}{3}$.
- s *value* number of range cell samples for hard weight computation. This refers to the number of range samples from each reference CPI to be used in the matrix on which a QR factorization is performed for hard weight computation. Figures 3.2 and 3.3 shows the QR factorization used in hard weight computation, given 16 channels and 3 reference CPIs. The suggested default value is 39. For best performance the value should be chosen to utilize the entire machine cache such that the whole QR factorization matrix can fit in the cache.
- b *value* number of receive beams for each broad transmit beam.
- g *value* number of range segments for hard Doppler bins. The default value of the number of hard segments is 6. One weight vector is computed for each hard segment. The user may increase the number of hard segments when needed for improved clutter cancelation performance at the expense of increasing computation time.
- l *value* number of broad transmit beams.
- d *value* the broad transmit beam direction. This value should be from 0, 1, ... (number of broad transmit beams -1).
- V *filename* filename for steering vector (the filename without suffix .mat). This file must be in Matlab 4.0 format. The size of the steering vector should be (number of broad transmit beams * number of receive beams * number of channels).

- c value* beam constraint weight. To preserve the main beam shape, the beam constraint weight is chosen to be large. To emphasize clutter cancelation at the expense of beam shape, the beam constraint weight is chosen to be small. Figures 3.1, 3.2, and 3.3 show this constant used in QR factorizations in both easy and hard weight computation, given 16 channels and 3 reference CPIs. The suggested default value is 0.5.
- f value* frequency constrain weight. Frequency constraints are included to reflect the desire for gain in the Doppler bin of interest. Large values of the frequency constraint weight result in large gain. Figures 3.2 and 3.3 show this constant used in QR factorizations in hard weight computation. The default value is 0.05.
- o value* forgetting factor used in QR factorization. Data from several reference CPIs are used. The forgetting factor is employed to diminish importance of the older data. Figure 3.3 shows this constant used in QR factorizations only in the successive steps of hard weight computation. The default value is 0.6.
- C filename* filename for replica complex array used in pulse compression (the filename without suffix .mat). This file must be in Matlab 4.0 format. The replica array must be of size the least number of power of 2 that is greater than or equals to number of range cells.
- a value* number of guard cells on each side of a test cell. The sliding window constant false alarm rate (CFAR) processing compares the value of a test cell at a given range to the average of a set of reference cells that surround it times a false alarm factor. The default value is 2.
- i value* number of reference cells to each side of the test cell is called the window size. The default value is 10.
- q value* the false alarm factor, a threshold scaler used to control the number of false alarms. Increasing this factor reduces the number of false alarms at the expense of target detections. The default value is 12.7
- v value* probability of false alarm for order statistic CFAR
- y value* guessing left boundary root of solving threshold equation
- z value* guessing right boundary root of solving threshold equation
- x value* accuracy of bisection root finding for solving threshold

- t *value* order number for order statistic CFAR
- R *value* recording start range (in meters)
- S *value* A/D sampling frequency (in Hz)
- N *value* number of bits to represent one CPI element
- P *value* transmit pulse width (in seconds)
- F *value* transmit frequency (in Hz)
- B *value* transmit bandwidth (in Hz)
- D *value* azimuth element spacing (in meters)
- A *value* mechanical boresight azimuth (degree)
- E *value* mechanical boresight elevation (degree)
- T to run in two threads in SMP system.

The required parameters are -k -j -n -m -r -b -l -d -V -R -S -N -P -F -B -D -A -E. An example for weight computation of using 128 pulses, 16 channels, 26 easy range samples, 39 hard range samples, 3 reference CPIs, and 6 receive beams is shown in Figure 3.1, 3.2, and 3.3.

3.9 Results output

The resulted output consists of the output of detected target report and the performance timing report. The output of detected target report is given in file `cfar_out.mat`, if the debugging flag `DEBUG_CFAR_PRINT_RESULT` is set in file `cfar_grp.c`.

The timing results are stored in the file `timing`. This file gives the communication and computation time for each of 7 tasks and the measured latency (in seconds) and throughput (in number of CPI data cubes) as well. An example of timing output is

```
total processors = 118
                nop    recv    comp    send    total
-----
Doppler filter & 16 & 0.0110 & 0.1714 & 0.0668 & 0.2492
```

```

easy weight      &   8 & 0.0998 & 0.1636 & 0.0003 & 0.2637
hard weight      &  56 & 0.0979 & 0.1636 & 0.0005 & 0.2621
easy BF          &   8 & 0.1302 & 0.1267 & 0.0036 & 0.2605
hard BF          &  14 & 0.1782 & 0.0822 & 0.0017 & 0.2622
pulse compr      &   8 & 0.1027 & 0.1543 & 0.0051 & 0.2621
CFAR detection &   8 & 0.1742 & 0.0864 & 0.0000 & 0.2606
Estimated throughput = 3.7919
Estimated Latency    = 1.0342 -----
Measured throughput  = 3.7959
Measured latency     = 0.6805

```

3.10 Debugging

Several defined macros are used for debugging purposes. They are designed for read from and write to Matlab files. The Matlab files used in this code are restricted for Matlab 4.x only (because Matlab 5.x and above have some headers at the beginning of the .mat files.) These macros only exist in the main subroutines of 7 tasks. Therefore, debugging can be done for each individual task by checking its input and output data files. Table 3.2 gives the names of these macros in each main subroutines.

Table 3.2: *Debugging macros for file main subroutines of 7 tasks.*

subroutine	macro	purpose
filter_grp.c	DEBUG_FILTER_READ_CPI_MAT DEBUG_FILTER_PRINT_RECV_CPI DEBUG_FILTER_PRINT_RESULT	read CPI from Matlab files print received CPI to Matlab files print result filtered CPI to Matlab files
wt_easy_grp.c	DEBUG_WT_EASY_PRINT_RECV_CPI DEBUG_WT_EASY_READ_CPI DEBUG_WT_EASY_PRINT_RESULT	print received CPI to a Matlab file read CPI samples from a Matlab file print result weight vectors to a Matlab file
wt_hard_grp.c	DEBUG_WT_HARD_PRINT_RECV_CPI DEBUG_WT_HARD_READ_CPI DEBUG_WT_HARD_PRINT_RESULT	print received CPI to a Matlab file read CPI samples from a Matlab file print result weight vectors to a Matlab file
bf_easy_grp.c	DEBUG_BF_EASY_PRINT_RECV_CPI DEBUG_BF_EASY_PRINT_RECV_WTS DEBUG_BF_EASY_READ_CPI DEBUG_BF_EASY_READ_WTS DEBUG_BF_EASY_PRINT_RESULT	print received CPI to a Matlab file print received weight vectors to a Matlab file read CPI from a Matlab file read weight vectors from a Matlab file print result beamformed data to a Matlab file
bf_hard_grp.c	DEBUG_BF_HARD_PRINT_RECV_CPI DEBUG_BF_HARD_PRINT_RECV_WTS DEBUG_BF_HARD_READ_CPI DEBUG_BF_HARD_READ_WTS DEBUG_BF_HARD_PRINT_RESULT	print received CPI to a Matlab file print received weight vectors to a Matlab file read CPI from a Matlab files read weight vectors from a Matlab file print result beamformed data to a Matlab file
pc_grp.c	DEBUG_PC_PRINT_RECV_BF DEBUG_PC_READ_BF DEBUG_PC_PRINT_RESULT	print received beamformed data to a Matlab file read beamformed data from a Matlab file print result pulse compression data to a Matlab file
cfar_grp.c	DEBUG_CFAR_PRINT_RECV_PC DEBUG_CFAR_READ_PC DEBUG_CFAR_PRINT_RESULT	print received CPI to a Matlab file read pulse compressed data from a Matlab file print result CFAR detection data to a Matlab file

Chapter 4

Examples of Running Codes

This chapter gives three examples of running parallel pipeline STAP codes, one for each of Intel Paragon, IBM SP, and SGI Origin machines. All input parameters and data files given here are the same across these three machines. The only difference are the compiling and executing commands. Users can use the following examples to run the program and obtain the same output results to make sure the proper use of the program.

The source code package has already been set to this example as the default options. Users can compile and execute these codes without modifying anything and check with the output results shown in this chapter.

4.1 Parameter files

There are two parameter files: `proc.dat` and `param.dat`. Node assignments to the tasks in the STAP pipeline system is given in file `proc.dat`. File `param.dat` provides all parameters that are relative to the signal processing.

The example of the file `proc.dat`:

```
% cat proc.dat
8 2 28 4 4 4 2
```

The example of the file `param.dat`:

```
% cat param.dat
-k      512      # number of range cells
-j      16       # number of channels
```

```

-n      128      # number of pulses
-r      3        # number of reference CPIs
-m      24        # total number of CPIs (besides the reference CPIs)
-p      3        # number of zero padding
-w Hanning      # windowing function: Hanning or Hamming
-h      56        # number of hard Doppler bins
-e      26        # number of range samples for easy weight
-u      0.3333    # fraction of range cells for extracting easy weight samples
-s      39        # number of range samples for hard weight
-g      6         # number of segments for each hard Doppler bin
-l      5         # number of broad transmit beams
-d      2         # broad transmit beams direction
-b      6         # number of receive beams for each broad transmit beam
-V      SVs       # filename of the steering vector (in Matlabs 4.0)
-c      0.5       # beam constraint weight
-f      0.05      # frequency constraint weight
-o      0.6       # forgetting factor
-C replica      # filename for replica array used in pulse compression
-a      2         # number of guard cells for the sliding window
-i      10        # number of range cells for the window size
-q      12.7      # false alarm factor
-v      0.0001    # probability of false alarm for order statistic CFAR
-y      0.0       # guessing left boundary root of solving threshold equation
-z      100.0     # guessing right boundary root of solving threshold equation
-x      0.0001    # accuracy of bisection root finding for solving threshold
-t      17        # order number for order statistic CFAR
-R      17400     # recording start range (in meters)
-S      1.0E6     # A/D sampling frequency (in Hz)
-N      16        # number of bits representing one CPI element
-P      61.1E-6   # transmit pulse width (in seconds)
-F      450.0E6   # transmit frequency (in Hz)
-B      0.5E6     # transmit bandwidth (in Hz)
-D      0.333     # azimuth element spacing (in meters)
-A      90        # mechanical boresight azimuth (degree)
-E      3         # mechanical boresight elevation (degree)

```

4.2 CPI data files

The default setting is using random generated CPI data as program input. The output of the program only shows the performance timing results. However, four CPI data files are also provided for the code testing: `cpi0.mat`, `cpi1.mat`, `cpi2.mat`, and `cpi3.mat`. These files are in Matlab 4.0 format. If users would like to see the target detection report of these four CPI data, two define macros have to be set:

- `DEBUG_FILTER_READ_CPI_MAT` in file `filter_grp.c` and file `filter_grp_np.c`
- `DEBUG_CFAR_PRINT_RESULT` in file `cfar_grp.c` and `cfar_grp_nt.c`.

Also, the total number of CPI data sets has to be set to 1 in the file `param.dat`, i.e.,

```
-m      1      # total number of CPIs (besides the reference CPIs)
```

The results of target report will be saved as a Matlab file, `cfar_out.mat`. Please refer to Section 4.5 to see the target output for using these 4 CPI files.

Otherwise, users can test the program for observing the performance results by doing the following:

1. not setting any define macros in all files
2. set the number of CPI data sets to any number larger than 20.

In this case, the program generates random numbers for CPI data as inputs to the STAP pipeline system. This will eliminate the overhead of reading and writing Matlab files into the disk. In this way, the performance timing results contain purely computation and communication costs on the parallel machines.

4.3 Compiling

The multi-threading implementation only works on Intel Paragon with SMP nodes. Therefore, there are two choices to compile the source codes on Paragon. To obtain the multi-threaded execution codes on Intel Paragon, use the command

```
% make -f Makefile.paragon
```

a multi-threaded execution code, `main_f`, will be generated. To obtain the single-threaded execution codes on Intel Paragon, use the command

```
% make -f Makefile.paragon nt
```

a single-threaded execution code, `main_nt`, will be generated.

On the IBM SP and SGI Origin, only single-threaded implementation work even if the multi-threaded code is compiled. To compile the source codes on IBM SP, use the command

```
% make -f Makefile.sp
```

to obtain a multi-threaded execution code `main_f` or

```
% make -f Makefile.sp nt
```

to obtain a single-threaded execution code `main_nt` To compile the source codes on SGI Origin, use the command

```
% make -f Makefile.sgi
```

to obtain a multi-threaded execution code `main_f` or

```
% make -f Makefile.sgi nt
```

to obtain a single-threaded execution code `main_nt`.

4.4 Executing

The following commands are for running the codes interactively. On Intel Paragon,

```
% main_nt -sz 52 -pn OPEN.stap
```

assuming the partition `.compute.OPEN.stap` has been built. On IBM SP,

```
% mpirun -np 52 main_nt
```

On SGI Origin,

```
% mpirun -np 52 main_nt
```

To run the program in the batch mode on Paragon, the command

```
% qsub -eo -o $HOME/STAP/q_out -q q512 -lP 52 -lT 20:00 -x scpt_file
```

submits a batch job requiring 52 processors, maximum 20 minutes, script file name `scpt_file` on the batch queue named `q512`. The script file, `scpt_file`, is

```
% cat scpt_file
$HOME/STAP/main_nt
```

To submit a batch job on IBM SP, use the command

```
% spsubmit -np 52 -progtype M -maxtime 20 -stdout q_out main_nt
```

and it will submit a batch job requiring 52 processors and maximum 20 minutes execution time and standard output to the file `q_out`.

4.5 Output

The performance timing results are given in file `timing`. An example of output using 52 nodes on the Intel Paragon is

```
total processors = 52

-----
          nop      recv      comp      send      total
-----
Doppler filter &   8 & 0.0213 & 0.2670 & 0.1349 & 0.4232
easy weight      &   2 & 0.0897 & 0.3331 & 0.0003 & 0.4231
hard weight      &  28 & 0.1364 & 0.2851 & 0.0003 & 0.4218
easy BF          &   4 & 0.2752 & 0.1433 & 0.0003 & 0.4188
hard BF          &   4 & 0.2360 & 0.1754 & 0.0003 & 0.4117
pulse compr      &   4 & 0.1839 & 0.1973 & 0.0293 & 0.4106
CFAR              &   2 & 0.2740 & 0.1363 & 0.0000 & 0.4103
Estimated throughput = 2.3630
Estimated Latency    = 1.6628 -----
Measured throughput  = 2.3865
Measured latency     = 1.0766
```

An example result on IBM SP is


```

total processors = 52
      nop      recv      comp      send      total
-----
Doppler filter & 8 & 0.0068 & 0.0593 & 0.0964 & 0.1625
easy weight    & 2 & 0.1208 & 0.0525 & 0.0001 & 0.1734
hard weight    & 28 & 0.1048 & 0.0639 & 0.0001 & 0.1689
easy BF        & 4 & 0.1072 & 0.0605 & 0.0001 & 0.1678
hard BF        & 4 & 0.1069 & 0.0615 & 0.0002 & 0.1686
pulse compr    & 4 & 0.1146 & 0.0527 & 0.0001 & 0.1674
CFAR           & 2 & 0.1296 & 0.0402 & 0.0000 & 0.1699
Estimated throughput = 5.7654
Estimated Latency    = 0.6684 -----
Measured throughput  = 5.9104
Measured latency     = 0.4273

```

Users can also test the output of target detection results when using provided 4 CPI Matlab data files. If the program is compiled to generate the target detection output, there are one output shows total number possible targets and one out Matlab file stores the target detection report. When total number of CPIs (besides the reference CPIs) is set to one, the output should be

Total number of possible targets = 148

There is also a target report file, `cfar_out.mat`, generated in Matlab format. This file can be loaded into Matlab software and printed into a file by running a Matlab subroutine name `tg_rpt()` provided in directory MAT. The results specified 148 targets each with three elements: Doppler bin number, receive beam number, and range cell number.

% matlab

< M A T L A B (R) >

(c) Copyright 1984-98 The MathWorks, Inc.

All Rights Reserved

Version 5.2.0.3084

Jan 17 1998

To get started, type one of these: `helpwin`, `helpdesk`, or `demo`.

For product information, type `tour` or visit www.mathworks.com.

```
>> load cfar_out;  
>> tg_rpt('default_target.txt', cfar_out, 128, 6, 450);  
>> quit
```

1036948 flops.

```
% cat default_target.txt
```

Doppler bin #	recv beam #	range #
-----	-----	-----
6	1	136
6	2	320
7	2	193
7	3	193
7	4	193
8	1	44
8	1	45
13	5	276
16	5	170
18	2	234
18	3	234
18	4	234
19	6	139
20	1	364
20	6	139
20	6	140
23	2	282
24	5	52
27	4	300
32	3	365
32	3	366
32	4	277
41	1	335
42	1	173
42	5	208

44	4	174
45	5	67
46	4	67
49	1	122
49	2	122
53	1	364
53	6	164
55	2	429
55	2	430
57	1	412
57	2	412
58	3	216
58	3	217
58	4	216
58	4	217
61	4	342
67	6	89
70	6	58
71	2	392
71	6	445
72	2	325
72	2	326
78	5	108
78	5	109
79	3	253
80	1	204
80	1	205
80	2	204
80	2	205
80	3	204
80	3	205
80	4	204
80	4	205
80	5	204
80	5	205
81	1	203
81	1	204

81	1	205
81	1	206
81	2	203
81	2	204
81	2	205
81	2	206
81	3	203
81	3	204
81	3	205
81	3	206
81	4	147
81	4	203
81	4	204
81	4	205
81	4	206
81	5	203
81	5	204
81	5	205
81	5	206
81	6	204
81	6	205
82	1	203
82	1	204
82	1	205
82	1	206
82	2	203
82	2	204
82	2	205
82	2	206
82	3	203
82	3	204
82	3	205
82	3	206
82	4	203
82	4	204
82	4	205
82	4	206

82	5	203
82	5	204
82	5	205
82	5	206
82	6	204
82	6	205
83	1	204
83	1	205
83	2	204
83	2	205
83	3	203
83	3	204
83	3	205
83	4	203
83	4	204
83	4	205
83	4	338
83	5	203
83	5	204
83	5	205
92	6	136
92	6	322
96	4	92
107	2	122
112	1	211
112	2	211
112	2	282
112	2	283
112	3	282
112	3	283
112	4	283
118	1	216
118	3	445
118	4	306
119	2	386
119	2	387
119	3	386

119	3	387
121	2	405
121	2	406
121	3	405
121	3	406
124	1	359
125	4	142
125	5	142
126	3	403
126	5	402
126	6	402
127	6	436

total number of targets = 148

4.6 Script to run with defaults

Script files are included in the software package for users to run the STAP code using the defaults described in this Chapter. Three scripts files are available for three High Performance Computers: Intel Paragon at California Institute of Technology, IBM SP at Argonne National Laboratory, and SGI Origin at Northwestern University. The machine platforms are shown in Table 4.1.

The script files are:

- Paragon - script_paragon
- SP - script_sp
- Origin - script_origin

Users can run these script files on each of three machines to compile and run the code in one time by using command:

- Paragon - % sh script_paragon
- SP - % sh script_sp
- Origin - % sh script_origin

Table 4.1: System platforms

	AFRL Paragon	ANL IBM SP	NWU SGI Origin
CPU Type	i860 RISC	P2SC [†]	MIPS R10000
RAM (MByte)	64	256	1024
MFLOPS/proc	100	480	390
MHz /proc	40	120	195
No. nodes	232	80	8
No. proc/node	3	1	1
Execution mode	dedicate	dedicate	time share

[†]P2SC: Power 2 SuperScalar chip

Bibliography

- [1] M. Little and W. Berry. Real-Time Multi-Channel Airborne Radar Measurements. *IEEE National Radar Conference*, 1997.
- [2] A. Choudhary, W. Liao, D. Weiner, P. Varshney, R. Linderman, and M. Linderman. Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers. *International Parallel Processing Symposium*, 1998.
- [3] A. Choudhary and J. Patel. *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems*. Kluwer Academic Publishers, Boston, MA, 1990.
- [4] A. Choudhary and R. Ponnusamy. Run-Time Data Decomposition for Parallel Implementation of Image Processing and Computer Vision Tasks. *Journal of Concurrency, Practice and Experience*, 4(4):313-334, June 1992.
- [5] A. Choudhary and R. Ponnusamy. Parallel Implementation and Evaluation of a Motion Estimation System Algorithm using Several Data Decomposition Strategies. *Journal of Parallel and Distributed Computing*, 14:50-65, January 1992.
- [6] R. Thakur, A. Choudhary, and J. Ramanujam. Efficient Algorithms for Array Redistribution. *IEEE Trans. on Parallel and Distributed Systems*, 6(7):587-594, June 1996.
- [7] M. Berger and S. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Trans. on Computers*, 36(5):570-580, May 1987.
- [8] F. Berman and L. Snyder. On Mapping Parallel Algorithms into Parallel Architectures. *Journal of Parallel and Distributed Computing*, 4:439-458, 1987.
- [9] A. Choudhary, B. Narahari, D. Nicol, and R. Simha. Optimal Processor Assignment for Pipeline Computations. *IEEE Trans. on Parallel and Distributed Systems*, April 1994.
- [10] M. Linderman and R. Linderman. Real-Time STAP Demonstration on an Embedded High Performance Computer. *IEEE National Radar Conference*, 1997.

- [11] R. Brown and R. Linderman. Algorithm Development for an Airborne Real-Time STAP Demonstration. *IEEE National Radar Conference*, 1997.
- [12] M. Snir and et. al. *MPI The Complete Reference*. The MIT Press, 1995.
- [13] Kuck and Associates, Champaign, IL. *CLASSPACK Basic Math Library / C*, 1994.
- [14] Kuck and Associates, Champaign, IL. *CLASSPACK Signal Processing Library*, 1994.
- [15] IBM, http://www.austin.ibm.com/resource/aix_resource/sp_books/essl. *Engineering and Scientific Subroutine Library for AIX Guide and Reference*, 1997.
- [16] Silicon Graphics Inc., <http://www.sgi.com/software/scsl.html>. *SGI/CRAY Scientific Library*, 1998.
- [17] Intel Corporation. *Paragon System User's Guide*, April 1996.

***MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)***

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.